

目录

1 体验分布式 KV 存储系统 erraftkv	4
1.1 开篇	4
1.2 erraftkv 架构	4
1.2.1 核心组件	5
1.2.2 系统中有三种角色	5
1.2.3 请求处理流程	5
1.3 安装 Go 编译环境	6
1.4 编译构建 erraftkv	6
1.5 让系统跑起来，体验它！	6
1.5.1 可执行文件介绍	7
1.5.2 启动服务	7
2 Go 语言基础知识	9
2.1 Go 语言优点	9
2.2 切片	10
2.2.1 使用字面量列表创建	10
2.2.2 从已有数组创建切片	10
2.2.3 修改切片中的元素	12
2.2.4 切片底层结构	13
2.3 Goroutine	16
2.4 调度器	17
2.4.1 调度器的设计决策	17
2.4.2 Go 调度器模型	19
2.5 内存管理	24
2.5.1 内存管理架构概览	24
2.5.2 Resident set (常驻集)	24
2.5.3 mheap	24
2.5.4 mspan	25

2.5.5	mcentral	25
2.5.6	arena	26
2.5.7	mcache	26
2.5.8	Stack	26
2.5.9	内存分配流程概要	26
3	Raft 论文解读	27
3.1	Raft 概览	27
3.2	分布式系统中的脑裂	29
3.3	多数派协议	30
3.4	Raft 的日志结构	31
3.5	Raft 的状态转换	31
3.6	Leader 选举	32
3.7	日志复制	33
3.8	日志合并和快照发送	34
4	构建 Raft 库	36
4.1	核心数据结构设计	36
4.2	协程模型	38
4.3	Rpc 定义	39
4.3.1	Entry	39
4.3.2	RequestVote 相关	40
4.3.3	AppendEntries 相关	41
4.4	Leader 选举实现分析	42
4.5	日志复制实现分析	52
4.6	Raft 快照实现分析	60
4.7	Raft 如何应对脑裂	64
5	基于 Raft 库，实现简单的分布式 KV 系统	66
5.1	系统架构概览	66

5.2 对外接口定义	67
5.3 服务端核心实现分析	69
5.4 客户端实现介绍	74
6 Multi-Raft 设计与实现	75
6.1 设计思考	75
6.2 配置服务器实现分析	76
6.3 分片服务器实现分析	78
6.4 客户端实现分析	84
6.5 用新硬件加速持久化存储	88
6.5.1 持久内存介绍	88
6.5.2 试用持久内存设备	89
7 分布式事务初探	97
7.1 事务介绍	97
7.2 ACID	99
7.3 两阶段提交	99

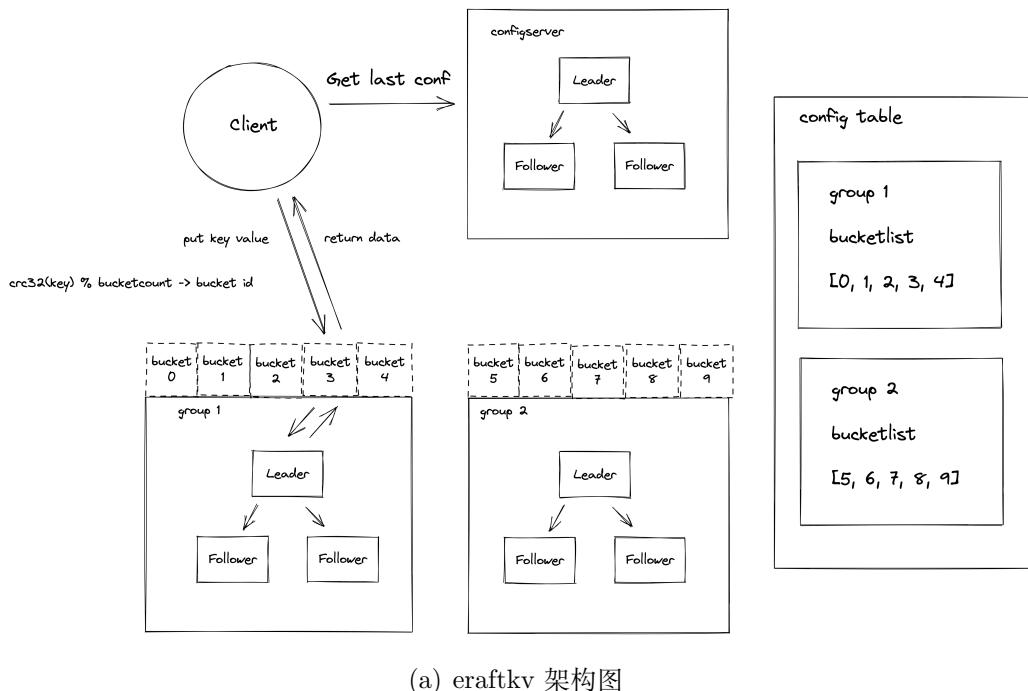
1 体验分布式 KV 存储系统 erraftkv

1.1 开篇

作为本书的开篇，我们首先会介绍一下分布式 KV 存储系统 - erraft 的架构，然后通过手动部署运行的方式带领大家直观感受分布式 KV 系统所具备的能力。

1.2 erraftkv 架构

在我们运行 erraftkv 之前，我们先熟悉一下它的架构。这有助于我们对 erraftkv 的工作流程有清晰的认识。



(a) erraftkv 架构图

上图是 erraftkv 的架构图，erraftkv 作为一个分布式 KV 存储系统，它的核心组件是 bucket 和 config table。erraftkv 的节点角色分为 Client , ConfigServer 和 ShardServer。不同的节点上部署的核心组件不同，我们是通过

节点上所部署的组件来判断节点的角色。接下来，我们将对 erraftkv 的核心组件和节点角色进行介绍。

1.2.1 核心组件

- 1.bucket - 它是集群中做数据管理的逻辑单元。
- 2.config table - 集群配置表。它主要维护了集群服务分组与 bucket 之间的映射关系。

1.2.2 系统中有三种角色

- 1.Client - 客户端。用户通过客户端来访问 erraftkv。
- 2.ConfigServer - 配置服务器。ConfigServer 是系统的配置管理中心。ConfigServer 上部署了 config table 组件，并存储了集群的路由配置信息。客户端在访问数据之前，要通过 ConfigServer 查询到 bucket 所在的服务分组列表，然后再访问数据。
- 3.ShardServer - 数据服务器。ShardServer 是系统中实际存储用户数据的服务器。ShardServer 部署了 bucket 组件。ShardServer 可以负责多个 bucket 的数据。

1.2.3 请求处理流程

在熟悉了 erraftkv 的架构之后，我们将通过分析一个具体的请求示例，来熟悉 erraftkv 的工作流程。

客户端向 erraftkv 集群发送一个 put testkey testvalue 请求：

1. 在启动时，客户端会从 ConfigServer 获取到最新的路由信息表和集群配置表。
2. 首先，客户端会计算 key 值“testkey”的 CRC32 哈希值，然后对集群中的 bucket 数取模，计算出 key 值命中哪个 bucket。
3. 然后，客户端将 put 请求内容打包成一个 RPC 请求包，并根据集群配置表的信息，将 RPC 请求发送到负责相应 bucket 的 ShardServer 服务

分组。

4. Leader ShardServer 节点接收到 RPC 请求后，并不是直接将数据写入存储引擎，而是构造一个 Raft 提案，然后将提案提交到 Raft 状态机中。当 ShardServer 分组中半数以上节点都同意这个操作后（如果没有接触过分布式一致性算法，这里可以先不用理解细节。我们会在第 4 章 Raft 论文解读中详细解读为什么要这样做），作为 Leader 的 ShardServer 节点才能给客户端返回写入成功。

1.3 安装 Go 编译环境

首先，我们需要在电脑上安装好 Go 语言编译器。你可以在 <https://go.dev/dl/> 官网下载对应你系统版本的安装包。按指示 <https://go.dev/doc/install> 安装 golang 编译环境。

1.4 编译构建 eraftkv

安装完 Go 语言编译器后，我们还需要安装 git 和 make 等基础工具。之后，我们按照如下命令编译 eraftkv

```
1 git clone https://github.com/eraft-io/eraft.git  
2 cd eraft  
3 make
```

1.5 让系统跑起来，体验它！

我们构建完 eraftkv 之后，在 eraft 目录有一个 output 文件夹，里面有我们需要运行的 bin 文件。

```
1 colin@B-M1-0045 eraft % ls -l output  
2 total 124120
```

```
3 -rwxr-xr-x 1 colin staff 12119296 5 25 20:40 bench_cli  
4 -rwxr-xr-x 1 colin staff 12114784 5 25 20:40 cfgcli  
5 -rwxr-xr-x 1 colin staff 13578848 5 25 20:40 cfgserver  
6 -rwxr-xr-x 1 colin staff 12127328 5 25 20:40 shardcli  
7 -rwxr-xr-x 1 colin staff 13600368 5 25 20:40 shardserver
```

1.5.1 可执行文件介绍

1.cfgserver

ConfigServer 的可执行文件，系统的配置管理中心，需要首先启动

2.cfgcli

ConfigServer 的客户端工具，它和 ConfigServer 交互用来管理集群的配置

3.shardserver

ShardServer 的可执行文件，它负责存储用户的数据

4.shardcli

ShardServer 的客户端工具，用户可以使用它向集群中写入数据

5.bench_cli

系统的性能测试工具

1.5.2 启动服务

1. 启动配置服务器分组

```
1 ./cfgserver 0 127.0.0.1:8088,127.0.0.1:8089,127.0.0.1:8090  
2  
3 ./cfgserver 1 127.0.0.1:8088,127.0.0.1:8089,127.0.0.1:8090  
4  
5 ./cfgserver 2 127.0.0.1:8088,127.0.0.1:8089,127.0.0.1:8090
```

2. 初始化集群配置

```
1  
2 ./cfgcli 127.0.0.1:8088,127.0.0.1:8089,127.0.0.1:8090  
3 join 1 127.0.0.1:6088,127.0.0.1:6089,127.0.0.1:6090  
4  
5 ./cfgcli 127.0.0.1:8088,127.0.0.1:8089,127.0.0.1:8090  
6 join 2 127.0.0.1:7088,127.0.0.1:7089,127.0.0.1:7090  
7  
8 ./cfgcli 127.0.0.1:8088,127.0.0.1:8089,127.0.0.1:8090  
9 move 0-4 1  
10  
11 ./cfgcli 127.0.0.1:8088,127.0.0.1:8089,127.0.0.1:8090  
12 move 5-9 2
```

3. 启动数据服务器分组

```
1  
2 ./shardserver 0 1 127.0.0.1:8088,127.0.0.1:8089,127.0.0.1:8090  
3 127.0.0.1:6088,127.0.0.1:6089,127.0.0.1:6090  
4  
5 ./shardserver 1 1 127.0.0.1:8088,127.0.0.1:8089,127.0.0.1:8090  
6 127.0.0.1:6088,127.0.0.1:6089,127.0.0.1:6090  
7  
8 ./shardserver 2 1 127.0.0.1:8088,127.0.0.1:8089,127.0.0.1:8090  
9 127.0.0.1:6088,127.0.0.1:6089,127.0.0.1:6090  
10  
11  
12 ./shardserver 0 2 127.0.0.1:8088,127.0.0.1:8089,127.0.0.1:8090  
13 127.0.0.1:7088,127.0.0.1:7089,127.0.0.1:7090
```

```
14  
15 ./shardserver 1 2 127.0.0.1:8088,127.0.0.1:8089,127.0.0.1:8090  
16 127.0.0.1:7088,127.0.0.1:7089,127.0.0.1:7090  
17  
18 ./shardserver 2 2 127.0.0.1:8088,127.0.0.1:8089,127.0.0.1:8090  
19 127.0.0.1:7088,127.0.0.1:7089,127.0.0.1:7090
```

4. 读写数据

```
1  
2 ./shardcli 127.0.0.1:8088,127.0.0.1:8089,127.0.0.1:8090 put  
3 testkey testvalue  
4  
5 ./shardcli 127.0.0.1:8088,127.0.0.1:8089,127.0.0.1:8090 get  
6 testkey
```

5. 运行基准测试

```
1  
2 ./bench_cli 127.0.0.1:8088,127.0.0.1:8089,127.0.0.1:8090 100  
3 put
```

2 Go 语言基础知识

2.1 Go 语言优点

Go 是一门开源的语言，它出生名门 Google，Go 语言社区有强有力的顶级技术人员支撑。Go 语言的设计者是 Rob Pike, Robert Griesemer, Ken Thompson。我们可以在这个页面找到他们的资料 <https://golang.design/history/>，其中 Ken Thompson 是 UNIX 系统的发明者。

Go 语言对于初学者很友好，很容易在短时间内快速上手。我们可以通过这个网站上提供的代码示例快速的入门 Go 语言：<https://gobyexample.com>。

作为多核时代的语言，Go 语言在设计之初就对并发编程有很多内置的支持，提供了极其健壮的相关标准库。利用这些标准库，我们可以快速地编写出高并发的应用程序。

国内外很多大厂都在使用 Go 语言。在 Go 语言强大的社区中，全世界优秀的技术人员开发了丰富的工具生态，有很多脚手架可以帮我们快速地构建应用程序。

2.2 切片

<https://www.callicoder.com/golang-slices/>

切片是一个数组的一段，基于数组构建，并提供了更多丰富的数据操作功能。开发者可以灵活和便利地操作数组。

在 Go 语言内部，切片只是对底层数组数据结构的引用。接下来我们将熟悉如何创建和使用切片，并了解它底层是怎么工作的。

2.2.1 使用字面量列表创建

这种方式类似 c++ 里面的初始化列表

```
1 var s = []int{3, 5, 7, 9, 11, 13, 17}
```

这种方式创建切片的时候，它首先会创建一个数组，然后返回对该数组切片的引用。

2.2.2 从已有数组创建切片

```
1 // Obtaining a slice from an array `a`  
2 a[low:high]
```

这里我们对 a 数组进行切割得到切片。这个操作得到的结果是索引 [low, high) 的元素，生成的切片包括索引低，但是不包括索引高之间的所有数组元素。

你可以运行下面的示例，理解这个切片操作

示例 1:

```
1 package main
2 import "fmt"
3
4 func main() {
5     var a = [5]string{"Alpha", "Beta", "Gamma", "Delta", "Epsilon"}
6
7     // Creating a slice from the array
8     var s []string = a[1:4]
9
10    fmt.Println("Array a = ", a)
11    fmt.Println("Slice s = ", s)
12}
13
14 // output
15
16 Array a = [Alpha Beta Gamma Delta Epsilon]
17 Slice s = [Beta Gamma Delta]
```

示例 2:

```
1 # Output
2 Array a = [C C++ Java Python Go]
3 slice1 = [C++ Java Python]
```

```
4 slice2 = [C C++ Java]
5 slice3 = [Java Python Go]
6 slice4 = [C C++ Java Python Go]
```

2.2.3 修改切片中的元素

切片是引用类型，它指向了底层的数组。当我们使用切片引用去修改数组中对应的元素的时候，引用相同数组的其他切片对象也会看到这个修改结果。

```
1 package main
2 import "fmt"
3
4 func main() {
5     a := [7]string{"Mon", "Tue", "Wed", "Thu", "Fri", "Sat",
6                     "Sun"}
7
8     slice1 := a[1:]
9     slice2 := a[3:]
10
11    fmt.Println("----- Before Modifications -----")
12    fmt.Println("a = ", a)
13    fmt.Println("slice1 = ", slice1)
14    fmt.Println("slice2 = ", slice2)
15
16    slice1[0] = "TUE"
17    slice1[1] = "WED"
18    slice1[2] = "THU"
```

```

19         slice2[1] = "FRIDAY"
20
21         fmt.Println("\n----- After Modifications -----")
22         fmt.Println("a = ", a)
23         fmt.Println("slice1 = ", slice1)
24         fmt.Println("slice2 = ", slice2)
25     }
26
27 // Output
28
29 ----- Before Modifications -----
30 a = [Mon Tue Wed Thu Fri Sat Sun]
31 slice1 = [Tue Wed Thu Fri Sat Sun]
32 slice2 = [Thu Fri Sat Sun]
33
34 type: post
35 ----- After Modifications -----
36 a = [Mon TUE WED THU FRIDAY Sat Sun]
37 slice1 = [TUE WED THU FRIDAY Sat Sun]
38 slice2 = [THU FRIDAY Sat Sun]

```

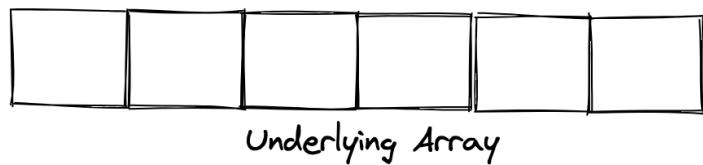
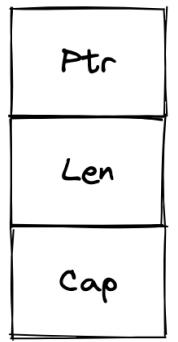
上面这个示例中，slice2 的修改操作在 slice1 中是能被看到的，slice1 的修改在 slice2 也能被看到。

2.2.4 切片底层结构

一个切片由三个部分组成，如图 b 所示

1. 一个指向底层数组的指针 Ptr
2. 切片所包含数组段的长度 Len
3. 切片的容量 Cap

Slice



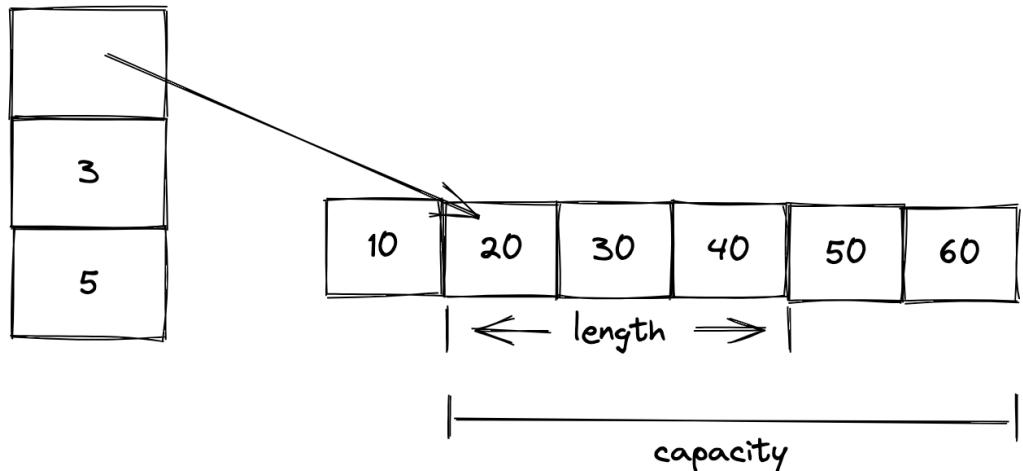
(a) Go 切片结构原理

我们看一个具体的切片结构的底层数例：

```
1 var a = [6]int{10, 20, 30, 40, 50, 60}  
2 var s = [1:4]
```

s 在 Go 内部是这样表示的：

Slice



(b) Go 切片示例

一个切片的长度和容量我们是可以通过 `len()`, `cap()` 函数获取的, 例如
我们可以通过下面的方式获取 s 的长度和容量。

```
1 package main
2 import "fmt"
3
4 func main() {
5     a := [6]int{10, 20, 30, 40, 50, 60}
6     s := a[1:4]
7
8     fmt.Printf("s = %v, len = %d, cap = %d\n", s, len(s),
9             cap(s))
10
11 // output
12 s = [20 30 40], len = 3, cap = 5
```

2.3 Goroutine

Goroutine 是由 Go 运行时所管理的一个轻量级的线程。一个 Go 程序中的 Goroutines 在相同的地址空间中运行，因此对共享内存的访问必须同步。

我们运行一个简单的示例来看看：

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func say(s string) {
9     for i := 0; i < 5; i++ {
10         time.Sleep(100 * time.Millisecond)
11         fmt.Println(s)
12     }
13 }
14
15 func main() {
16     go say("world")
17     say("hello")
18 }
19
20 // output
21
```

```
22 hello
23 world
24 hello
25 world
26 hello
27 world
28 world
29 hello
30 hello
```

我们可以看到主线程中 say("hello") 和 goroutine 的 say("world") 在交替的输出，它们在同时的运行。在其他语言做这个事情先要创建线程，然后绑定相关的执行函数，而 Go 语言直接把并发设计到了编译器语言支持层面，用 go 关键字就可以轻松地创建轻量级的线程。

2.4 调度器

2.4.1 调度器的设计决策

在解释 Go 语言调度器之前，我们先一个例子：

```
1
2 func main() {
3     var wg sync.WaitGroup
4     wg.Add(11)
5     for i := 0; i <= 10; i++ {
6         go func(i int) {
7             defer wg.Done()
8             fmt.Printf("loop i is - %d\n", i)
9         }(i)
10    }
```

```
11     wg.Wait()
12     fmt.Println("Hello, Welcome to Go")
13 }
14
15
16 // output
17
18 loop i is - 0
19 loop i is - 4
20 loop i is - 1
21 loop i is - 2
22 loop i is - 3
23 loop i is - 8
24 loop i is - 7
25 loop i is - 9
26 loop i is - 5
27 loop i is - 10
28 loop i is - 6
29 Hello, Welcome to Go
```

这个程序创建了 11 个 Goroutine，对于这个输出结果，我们可能会问：

(1) 这 11 个 Goroutine 是如何并行运行的？

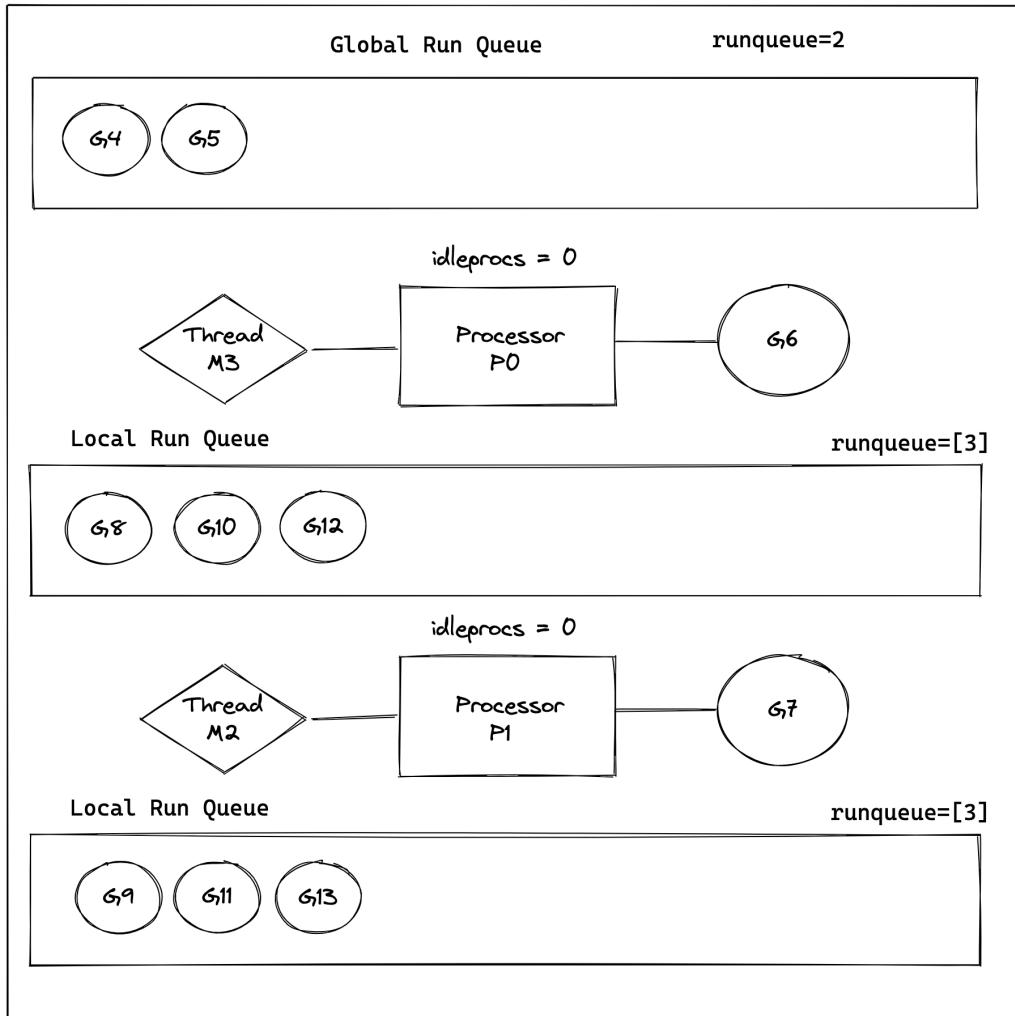
(2) 它们运行有没有特别的顺序？

要回答这两个问题，我们需要思考：

(1) 如何将多个 Goroutine 分配到 CPU 核心数量有限的机器上运行？

(2) 为了公平地让这些 Goroutine 获得 CPU 资源，这些 Goroutine 应该以什么的顺序在这多个 CPU 核心上运行？在下面的 Go 调度器模型章节，我们将回答上述问题。

2.4.2 Go 调度器模型



(c) Golang 调度器模型

为了解决上面的调度问题，Go 语言设计了图 c 中的调度器模型：
 Go 语言的调度模型叫做 GMP 模型，其中：
 G: 代表一个 Goroutine，是我们使用 go 关键字创建出来的可以并行运行的代码块。

M: 代表一个操作系统线程

P: 代表逻辑处理器

我们看到上图中由 2 个 P 在负责 8 个 Goroutine 的调度工作。

图中我们看到有两种类型的队列：

本地队列 (Local Run Queue): 存放等待运行的 G，这个队列存储的数量有限，一般不能超过 256 个，当用户新建 Goroutine 时，如果这个队列满了，Go 运行时会将一半的 G 移动到全局队列中。

全局队列 (Global Queue): 存放等待运行的 G，其他的本地队列满了，会移动 G 过来。

Go 调度器的工作流程

GMP 调度器调度 Goroutine 执行的大致逻辑如下：

1. 线程想要调度 G 执行就必须要先与某个 P 关联
2. 然后从 P 的本地队列中获取 G
3. 如果本地队列中没有可运行的 G 了，M 就会从全局队列拿一批 G 放到本地的 P 队列
4. 如果全局队列也没有可以运行的 G 的时候，M 会随机的从其他的 P 的本地队列偷一半 G 任务放到自己 (P) 的本地队列中。
5. 拿到可以运行的 G 之后，M 运行 G, G 执行完成之后，M 会运行下一个 G，一直重复执行下去。

跟踪 Go 调度器工作流程

Go 提供了 GODEBUG 工具可以跟踪调度器调度过程上述模型实时状态

我们使用下面的程序示例来追踪一下 Go 调度器是如何调度执行程序中的 Goroutine 的

```
1 package main
2
3 import (
4     "sync"
```

```
5         "time"
6     )
7
8 func main() {
9     var wg sync.WaitGroup
10    wg.Add(10)
11    for i := 0; i < 10; i++ {
12        go work(&wg)
13    }
14
15    wg.Wait()
16
17    // Wait to see the global run queue deplete.
18    time.Sleep(3 * time.Second)
19 }
20
21 func work(wg *sync.WaitGroup) {
22     time.Sleep(time.Second)
23
24     var counter int
25     for i := 0; i < 1e10; i++ {
26         counter++
27     }
28
29     wg.Done()
30 }
```

代码中创建了十个 Goroutine, 每个 Goroutine 都在做循环加 counter 值的操作。

我们编译上述例子

```
1 go build go_demo.go
```

然后使用 GODEBUG 工具来分析观察这些 Goroutine 的调度情况
执行命令：

```
1 GOMAXPROCS=2 GODEBUG=schedtrace=1000 ./go_demo
```

可以得到看到如下的输出，当然机器不一样可能输出会不一样，以下是在我的笔记本上输出的，我的本子有四个核心，下面的指令我指定了创建两个逻辑处理核心

```
1 colin@book % GOMAXPROCS=2 GODEBUG=schedtrace=1000 ./go_demo
2 SCHED 0ms: gomaxprocs=2 idleprocs=1 threads=4 spinningthreads=0
           idlethreads=1 runqueue=0 [0 0]
3 SCHED 1009ms: gomaxprocs=2 idleprocs=0 threads=4
               spinningthreads=0 idlethreads=1 runqueue=0 [8 0]
4 SCHED 2009ms: gomaxprocs=2 idleprocs=0 threads=4
               spinningthreads=0 idlethreads=1 runqueue=2 [3 3]
5 SCHED 3016ms: gomaxprocs=2 idleprocs=0 threads=4
               spinningthreads=0 idlethreads=1 runqueue=2 [3 3]
6 SCHED 4017ms: gomaxprocs=2 idleprocs=0 threads=4
               spinningthreads=0 idlethreads=1 runqueue=7 [0 1]
7 SCHED 5027ms: gomaxprocs=2 idleprocs=0 threads=4
               spinningthreads=0 idlethreads=1 runqueue=5 [0 3]
8 SCHED 6031ms: gomaxprocs=2 idleprocs=0 threads=4
               spinningthreads=0 idlethreads=1 runqueue=3 [2 3]
9 SCHED 7037ms: gomaxprocs=2 idleprocs=0 threads=4
               spinningthreads=0 idlethreads=1 runqueue=5 [1 2]
```

```
10 SCHED 8045ms: gomaxprocs=2 idleprocs=0 threads=4
    spinningthreads=0 idlethreads=1 runqueue=4 [2 2]
11 SCHED 9052ms: gomaxprocs=2 idleprocs=0 threads=4
    spinningthreads=0 idlethreads=1 runqueue=8 [0 0]
12 SCHED 10065ms: gomaxprocs=2 idleprocs=0 threads=4
    spinningthreads=0 idlethreads=1 runqueue=4 [0 4]
13 SCHED 11069ms: gomaxprocs=2 idleprocs=0 threads=4
    spinningthreads=0 idlethreads=1 runqueue=4 [1 3]
```

输出信息的含义如下

我们选取第二条分析

```
1 SCHED 1009ms: gomaxprocs=2 idleprocs=0 threads=4
    spinningthreads=0 idlethreads=1 runqueue=0 [8 0]
```

1009ms: 这个是从程序启动到这个 trace 采集度过的时间

gomaxprocs=2: 配置的逻辑处理核心，我们启动命令中写的

idleprocs=0: 空闲逻辑核心的数量

threads=4: 运行时正在管理的线程数量

idlethreads=1: 空闲线程的数量，这里有 1 个空闲，3 个正在运行中

runqueue=0: 全局运行队列中 Goroutine 的数量

[8 0]: 表示逻辑核心上本地队列中排队中 Goroutine 的数量，我们看到有一个核心上面有 8 个 Goroutine，另一个有 0 个，当然我们看后面的 trace 后续这两个核心的本地队列上都有任务了

了解更多调度器原理

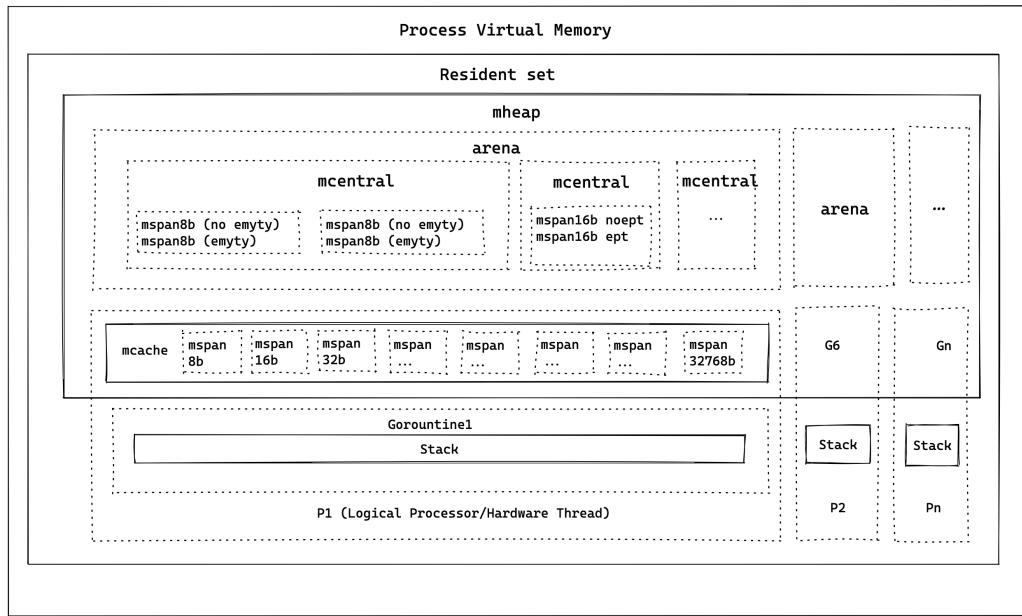
Go 语言是开源的，你可以在这个文件里面找到调度器的主要逻辑，<https://github.com/golang/go/blob/master/src/runtime/proc.go>，目前最新的代码有 6 千多行了，值得去读一读，读懂之后是很有收获的。

2.5 内存管理

2.5.1 内存管理架构概览

Go 最早的内存分配发源自 tcmalloc，它比普通的 malloc 性能要好，随着 Go 语言的不断演进，当前的内存管理性能已经非常好了。

我们首先通过图 d 来看 Go 内存管理架构图。



(d) Golang 内存管理架构图

图中涉及到的主要结构如下：

2.5.2 Resident set (常驻集)

虚拟内存划分为每个 8kb 的页面，由一个全局的 mheap 对象管理

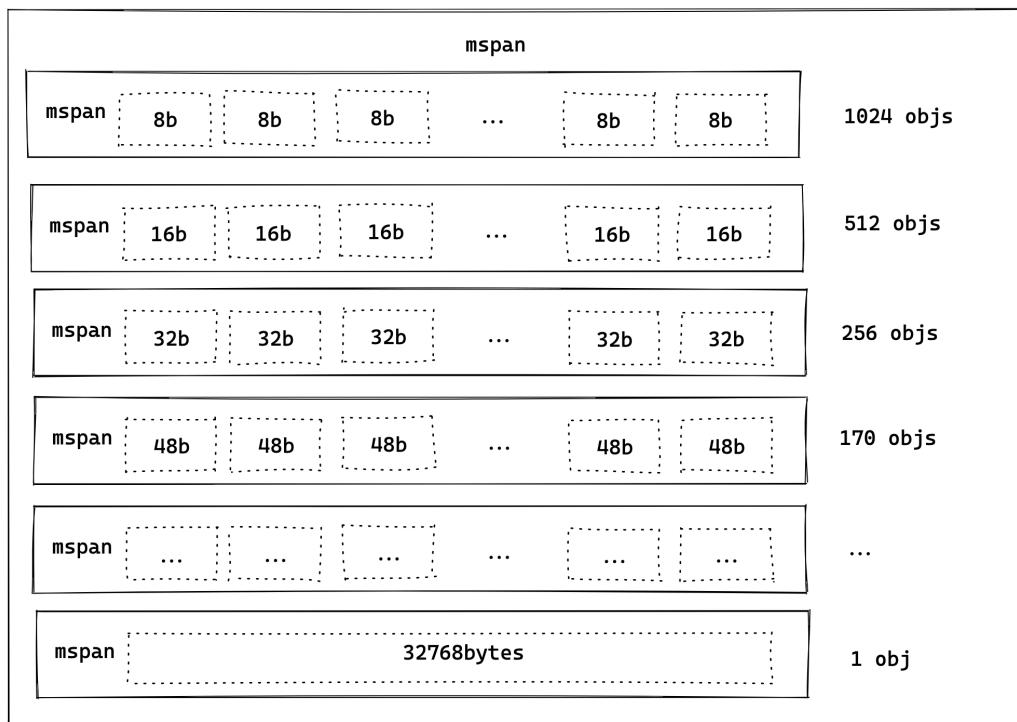
2.5.3 mheap

mheap 管理了 Go 语言动态存储的数据结构（即无法在编译时计算大小的数据）。mheap 是最大的内存块，也是 Go 垃圾回收发生的地方。

mheap 里面有管理了不同结构的页面，主要结构如下

2.5.4 mspan

mspan 是 mheap 中管理内存页的最基本结构，它底层结构是一个双向链表，span size class，以及 span 中的页面数量。和 tcmalloc 的操作一样，Go 将内存页面按大小划分为 67 个不同类的块，从 8b 到 32kb 不等，如图 e 所示



(e) Golang 内存管理 span

2.5.5 mcentral

mcentral 将相同大小 span 类组成分组，每个 mcentral 中包含两个 mspan：

`empty`: 一个双向的 span 链表, 其中没有空闲的对象或者 span 缓存在 mcache 中

`non-empty`: 有空闲对象的双链接列表, 当 mcentral 请求新的 span 的时候, 会从 `non-empty` 移动到 `empty` list

当 mcentral 没哟任何空闲的 span 是, 它会向 mheap 请求一个新的运行页面

2.5.6 arena

堆内存在分配的虚拟内存中根据需要进行扩大和收缩, 当需要更多的内存时, mheap 从虚拟内存中拉大小为 64MB 的内存块出来, 这个被叫做 arean。

2.5.7 mcache

mcache 是提供给 P (逻辑处理核心) 的内存缓存, 用来存储小对象 (也就是大小 $\leq 32\text{kb}$)。这有点类似于线程栈, 但是它其实是堆的一部分, 用于动态数据。mcache 中包含了 scan 和 noscan 类型所有大小的 mspan 对象

Goroutine 们可以从 mcache 中获取内存, 不需要加人任何锁, 因为 P 在同一时刻只能调度一个 G, 因此这是很高效的, mcache 在需要的时候会向 mcentral 中获取新的 span

2.5.8 Stack

这里是管理堆栈的内存区域, 每个 Goroutine 都有一个堆栈, 这里用来存储静态数据, 包括函数框架, 静态的结构, 原语值和指向动态数据机构的指针。

2.5.9 内存分配流程概要

分配器会按对象大小分配

`Tiny`

对于 Tiny (超小, size < 16 B) 对象:

直接使用 mcache 的 tiny 分配器分配大小小于 16 字节的对象，这是非常高效的。

Small

对于 Small (小型, size 16B ~ 32KB) 对象:

大小在 16 字节到 32 k 字节的对象分配时，在运行中 G 的 P 上的 mcache 里面获取。

在 Tiny 和 Small 分配中，如果 mspan 列表为空，没有页面用来分配了，分配器将从 mheap 获取一系列页面用于 mspan。

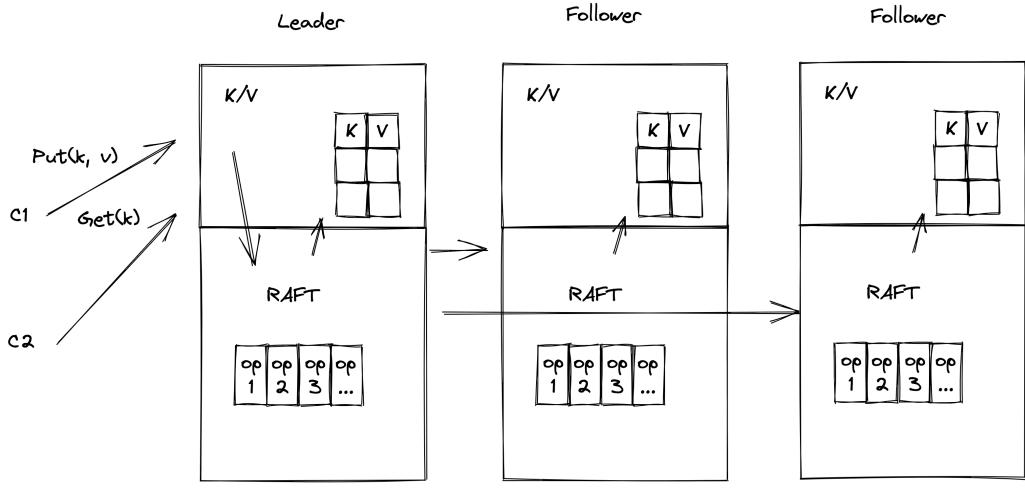
Large

对于 Large (大型, size > 32KB) 对象，直接分配在 mheap 相应大小的类上。如果 mheap 为空或者没有足够大的页面来分配了，那么它会从操作系统的进程虚拟内存分配一组新的页面过来（至少 1MB）。

3 Raft 论文解读

3.1 Raft 概览

这一小节我们不深入 Raft 算法细节，而是带着大家概览一下 Raft 算法在一个实际的应用系统中的应用。



(a) Golang raft 算法概览

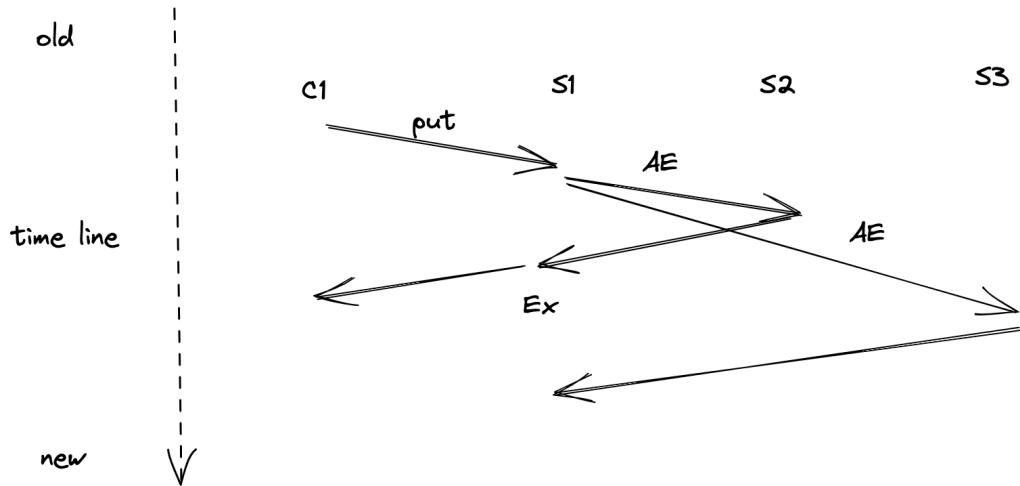
图 a 是一个使用 Raft 算法实现的一个分布式 KV 系统。我们这个系统的设计目标是保证集群中所有节点状态一致，也就是每个节点中 KV 表（这里使用通俗的“表”的概念描述，实际这些数据会存储到一个存储引擎里面）里面的数据状态最终是一致的。

先不考虑故障的场景，我们来看看系统在正常的情况下是怎么运行的。

我们来分析一下 Put 操作经过这个系统的流程。

1. 首先客户端会将 Put 请求发送给当前 Raft 集群中的 Leader 节点对应的 K/V 应用层。
2. Put 操作会被 Leader 包装成一个操作提交给 Raft 层，Raft 对这个 Put 请求生成一条日志存储到自己的日志序列中。
3. 同时，Raft 会把这个的操作日志，复制给集群中的 Follower 节点，当集群中的半数以上节点都复制这个日志并返回响应之后，Leader 会提交这条日志，并应用这条日志，写入数据到 KV 表，并通知应用层，这个操作成功执行。
4. 这时候 K/V 层会响应客户端，同时 Leader 会把 Commit 信息在下一次复制请求带给 Follower，Follower 也会应用这条日志，写入数据到 KV 表中，最终集群中所有节点的状态一致，整个系统的运行的时序如图 f 所示。

示。



(b) Golang raft kv 运行时序

这就是应用 Raft 保证系统一致性状态的例子。乍一看，像是很简单。但是当我们深入到算法细节里面的时候，这个系统就变复杂了。例如日志复制的时候会有很多约束条件来保证提交日志的一致性，以及在发生故障时如何正确的选出下一个 leader？多次故障之后，日志状态一致性如何保证？这些问题也是我们后续分析的重点。我们会结合具体代码，尽量简单地让我们理解 Raft 对这些问题的解决办法。

3.2 分布式系统中的脑裂

在我们介绍 Raft 算法之前我们先来看一下分布式的脑裂的问题。脑裂字面上是大脑裂开的意思，大脑是人体的控制中心，如果裂开了，那么整个系统就会出现紊乱。

对应到我们分布式系统里面，一般就是集群中的节点由于网络故障或者其他故障被划分成不同的分区。这个时候，由于不同分区无法通信，系统会出现状态不一致的情况。如果系统没有考虑处理这种情况，那么当网络再恢复的时候，系统也就没法再保证正确性了。

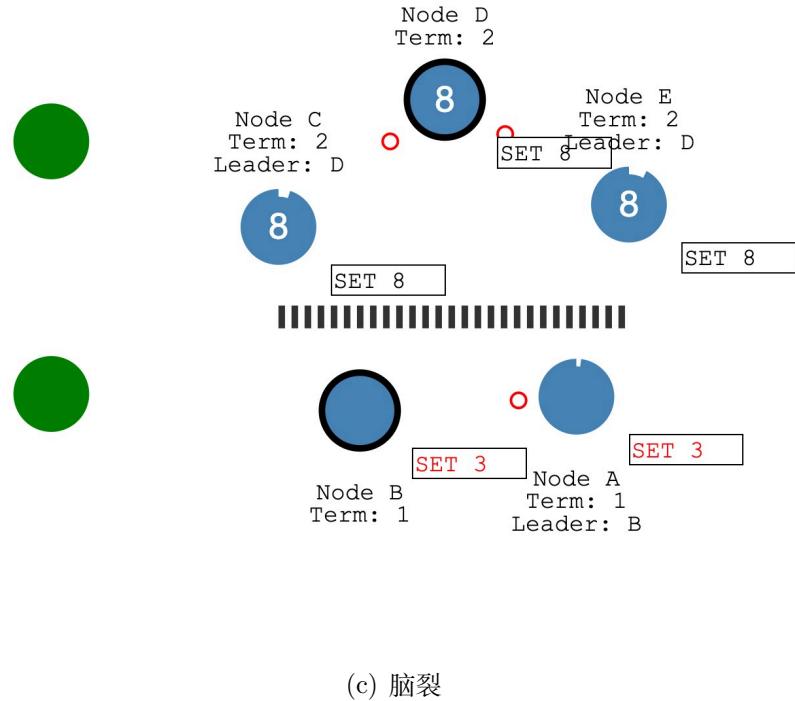


图 c 是分布式系统出现网络分区的情形。系统里面有 A-E 五个节点。由于故障，A,B 节点和 C,D,E 节点被划分到了各自的网络分区里面。绿色的圆形代表两个客户端，如果它们向不同的分区节点写入数据，那么系统能保证分区恢复后状态一致吗？Raft 算法解决了这个问题。在接下来的章节，我们将会详细讨论 Raft 是如何解决这个问题的。

3.3 多数派协议

Raft 论文中提到的半数票决 (Majority Vote)，也叫做多数派协议，是解决脑裂问题的关键。首先我们来解释一下半数票决是怎么做的，假设分布式系统中有 $2*f + 1$ 个服务器。系统在做决策的时候需要系统中半数节点以上投票同意，也就是必须要 $f + 1$ 个服务器都要活着，系统才能正常工作。那么这个系统最多可以接受 f 个服务器出现故障。

Raft 正是应用了半数票决来解决脑裂问题。假设我们的分布式系统由奇数个节点 ($3, 5 \dots 2n+1$) 个节点组成。一旦出现网络分区，那么必然会有任何一个分区存在半数节点以上的，那么过半票决这个策略就能正常运行。这样系统就不会因此不可用，多数派票决正是解决脑裂问题的关键。

3.4 Raft 的日志结构

前面我们概览了整个 Raft 算法的流程，请求经过系统最开始就要写入 Raft 算法层的日志。那这个日志的结构是什么样的呢？接下来我们就来了解 Raft 日志的结构：

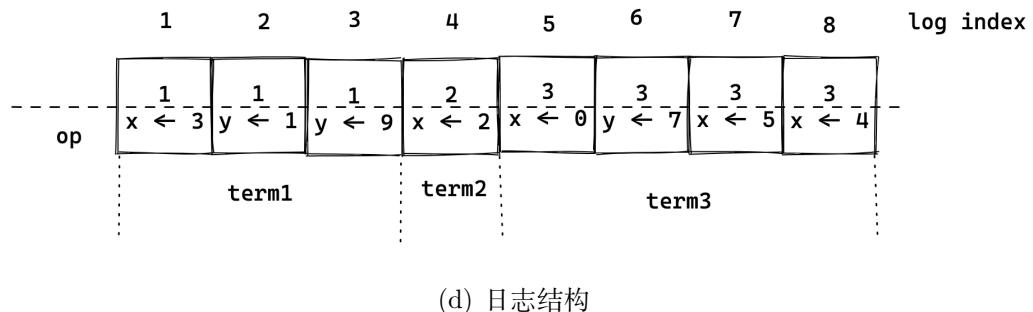
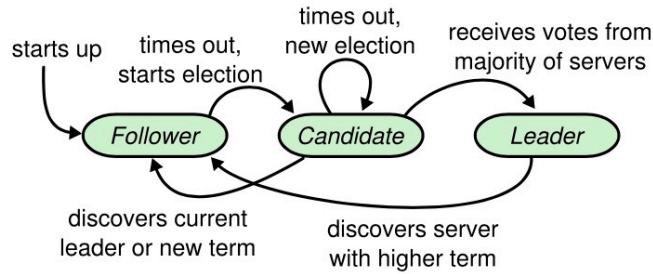


图 d 表示一个 Raft 节点日志的结构。日志主要用来记录用户的操作。我们会对这些操作进行编号。图中每一条（1~8）日志都有独立的编号 log index。此外，日志中还有任期号，如图中 1~3 号日志为任期 1 的日志。任期号是用来表示日志的选举状态。我们后面解释它的作用。每个日志都有一个操作。这个操作是对状态机的操作，如 1 号日志我们的操作是把 x 设置成 3。

3.5 Raft 的状态转换

Raft 协议的工作模式是一个 Leader 和多个 Follower 节点的模式。在 Raft 协议中，每个节点都维护了一个状态机。该状态机有 3 中状态：Leader、

Follower 和 Candidate。在系统运行的任意一个时间点，集群中的任何节点都处于这三个状态中的一个。



(e) Raft 状态转换

每个节点一启动就会进入 Follower 状态。当选举超时时间到达后，它会转换成 Candidate 状态。这时候该节点开始选举了。当该节点获得半数节点以上的选票之后，Candidate 状态的节点会转变成 Leader。或者当 Candidate 状态的节点发现了一个新的 Leader 或者收到新任期的消息，它会变成 Follower。Leader 发现更高任期的消息也会变成 Follower。在系统正常运行过程中，节点会一直在这三种状态之间转换。

3.6 Leader 选举

在介绍 Leader 选举流程之前，我们先来解释下 Raft 协议中与选举相关的两个超时时间：

选举超时（election timeout）时间和心跳超时（heartbeat timeout）时间。当 Follower 节点在选举超时时间之内没有收到来自 Leader 的心跳消息之后，就会切换成 Candidate 状态开始新一轮的选举。选举超时时间一般设置为 150ms ~ 300ms 的随机数。设置为随机数的目的是为了避免节点同时发起竞选，导致出现多个节点具有相同票数，从而选举失败重新选举的情况。提高选举超时时间的随机性有利于更快地选出 Leader。心跳超时

间则是指 Leader 向 Follower 节点发送的心跳消息的间隔时间。

我们来梳理一下选举的流程

1. 集群初始化，所有节点都会变成 Follower 状态。
2. 经过一段时间后（选举超时时间到达）Follower 还没收到来自 Leader 的心跳消息，那么它会开始切换为 Candidate 状态开始发起选举。
3. 变成 Candidate 之后节点的任期号也会增加，同时投给自己一票，然后并行地向集群中的其他节点发送请求投票（RequestVoteRPC）消息。
4. Candidate 节点赢得了半数以上选票后，该节点会成为 Leader 节点。之后该 Leader 节点会散播心跳消息给集群中其他节点，说明该节点成功选举 Leader。

以上是 Raft Leader 选举的大致流程，但是有两个细节需要注意：

1. 在等待投票的过程中，Candidate 可能会收到来自另外一个节点成为了 Leader 之后发送的心跳消息。如果这个消息中 Leader 的任期号（term）大于 Candidate 当前记录的任期号，Candidate 会认为这个 Leader 是合法的，它会变换为 Follower 节点。如果这个心跳消息的任期号小于 Candidate 当前的任期号，Candidate 将会拒绝这个消息，继续保持当前状态。
2. 另一种可能的结果是 Candidate 即没有赢得选举也没有输：也就是集群中多个 Follower 节点同时成为了 Candidate。这种情况叫做选票分裂，没有任何 Candidate 节点获得大多数选票。当这种情况发生的时候，每个 Candidate 会重新设置一个随机的选举超时时间，然后继续选举。由于选举时间是随机的，下一轮选举很大概率会有一个节点获得多数选票会成为新的 Leader。

3.7 日志复制

我们假设集群中有 A,B,C 三个节点，其中节点 A 为 Leader 节点，B,C 为 Follower 节点。此时客户端发送了一个操作到集群中：

1. 客户端请求首先将会发送给集群的 Leader 节点。收到请求消息后，Leader 节点 A 将会更新操作记录到本地的 Log 中。

2. Leader 节点 A 会向集群中其他节点发送 AppendEntries 消息。消息中记录了 Leader 节点 A 最近收到的客户端提交请求的日志信息（还没有同步给 Follower 的部分）。

3. 当 Follower 节点 B,C 收到来自 Leader 节点 A 的 AppendEntries 消息时，Follower 节点 B,C 会将操作记录到本地的 Log 中，并通知 Leader 成功追加日志的消息。

4. 当 Leader 节点 A 收到半数节点以上成功追加日志响应消息时，Leader 节点 A 会认为集群中有半数节点完成了日志同步操作，它会将日志提交的 committed 号更新。

5. Leader 节点 A 向客户端返回响应，并且在下一次发送 AppendEntries 消息的时候把 commit 号通知给 Follower 节点 B,C 。

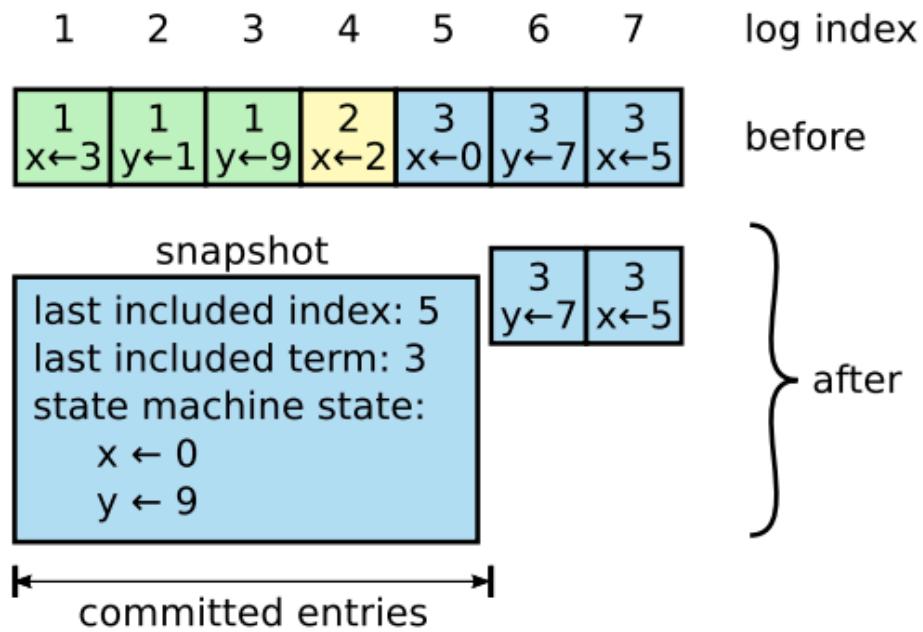
6. Follower 节点 B,C 收到消息之后，也会更新自己本地的 commit 号。

注意：上述流程是在正常情形下的流程。如果 Follower 节点宕机了或者运行很慢或者 Leader 节点发送的消息丢失了。Leader 上记录了到某个 Follower 同步的日志进度，如果追加请求没成功，会不停的重新发送消息，直到所有 Follower 都存储了所有的日志条目。

3.8 日志合并和快照发送

Raft 论文在第 7 章介绍了日志压缩合并相关要点。根据我们前面介绍的 Raft 复制相关内容，只要客户端有新的操作进来，Raft 就会写我们的日志文件，并且 Leader 同步给 Follower 之后，集群中所有节点的日志量都会随着操作的变多一直增长。raft 中使用 levelDB 存储了 Raft 日志条目。如果有节点挂了重新加入集群，我们需要给它追加大量的日志，这个操作会非常消耗 IO 资源，影响系统性能。

那么如何解决这个问题呢，首先我们在分析下日志结构：



(f) Raft 日志快照

从图 f 中，我们可以看到这里的操作，都是对 x, y 的操作。每次日志提交完，我们都会应用日志到状态机中。这里我们会发现其实我们并没有必要存储每一个日志条目。我们只关心一致的状态，也就是已经提交的日志让状态机最终到达一种什么样的状态。那么在图中，我们就可以把 1,2,3,4 号日志的操作之后状态机记录下来，也就是 $x <- 0$, $y <- 9$ ，并且记录这个状态之后第一条日志的信息，然后 1~4 号日志可以被安全地删除了。

以上的整个操作在 Raft 里面被叫做快照 (snapshot)。Raft 会定期的打快照把历史的状态记录下来。当集群中有某个节点挂了，并且日志完全无法找回之后时，集群中 Leader 节点首先会发送快照给这个挂掉之后新加入的节点，并且用一个 InstallSnapshot 的 RPC 发送快照数据给它。节点安装快照数据之后，会继续同步增量的日志，这样新的节点能快速的恢复状态。

4 构建 Raft 库

4.1 核心数据结构设计

我们上一章节讲了 Raft 算法的主要内容。这一章，我们将介绍它的具体实现。首先我们需要抽象出我们需要的数据结构。先来梳理一下可能用到的数据结构，首先节点之间需要互相访问，那我们需要定义访问其他节点的网络客户端。这里面要包含节点的 id，地址，还有 rpc 的客户端，整个结构我们抽象为 RaftClientEnd，主要数据内容如下：

```
1
2 type RaftClientEnd struct {
3     id          uint64
4     addr        string
5     raftServiceCli *raftpb.RaftServiceClient // grpc 客户端
6 }
```

节点状态我们之前描述的有三种，定义如下：

```
1
2 const (
3     NodeRoleFollower NodeRole = iota
4     NodeRoleCandidate
5     NodeRoleLeader
6 )
```

我们要完成选举操作的话需要两个超时时间，这里我们使用 Golang time 库里面的 Timer 实现，它可以定时的给一个通道发送消息，我们可以用它来实现选举超时和心跳超时。

```
1
```

```
2 electionTimer *time.Timer  
3 heartbeatTimer *time.Timer
```

此外，我们还构造了一个 Raft 结构体。Raft 结构体记录了当前节点的 id, 当前的任期号, 为谁投票, 获得票数的统计, 已经提交的日志索引号, 最后 apply 到状态机的日志号, 以及节点如果是 Leader 的话需要记录到其他节点复制最新匹配的日志号。Raft 结构体定义如下:

```
1  
2 type Raft struct {  
3     mu          sync.RWMutex  
4     peers       []*RaftClientEnd // rpc 客户端  
5     me_         int // 自己的 id  
6     dead        int32 // 节点的状态  
7     applyCh     chan *pb.ApplyMsg // apply 协程通道, 协程模  
    型中会讲到  
8     applyCond   *sync.Cond // apply 流程控制的信号量  
9     replicatorCond []*sync.Cond // 复制操作控制的信号量  
10    role        NodeRole // 节点当前的状态  
11    curTerm     int64 // 当前的任期  
12    votedFor    int64 // 为谁投票  
13    grantedVotes int // 已经获得的票数  
14    logs        *RaftLog // 日志信息  
15    commitIdx   int64 // 已经提交的最大的日志 id  
16    lastApplied  int64 // 已经 apply 的最大日志的 id  
17    nextIdx     []int // 到其他节点下一个匹配的日志 id 信  
    息  
18    matchIdx    []int // 到其他节点当前匹配的日志 id 信息  
19  
20    leaderId    int64 // 集群中当前 Leader 节点的 id
```

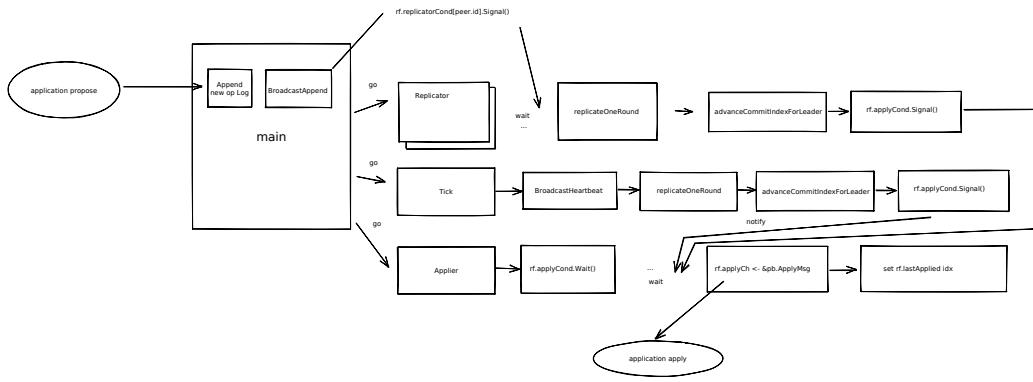
```

21     electionTimer *time.Timer // 选举超时定时器
22     heartbeatTimer *time.Timer // 心跳超时定时器
23     heartBeatTimeout uint64    // 心跳超时时间
24     baseElecTimeout uint64    // 选举超时时间
25 }

```

系统启动的时候，需要构造 Raft 结构体。这个流程是在 MakeRaft 里面实现，它主要是初始化变量和两个定时器，并启动相关的协程。我们这里对每个对端节点复制有 Replicator 协程，触发两个超时时间有 Tick 协程，应用已经提交的日志有 Applier 协程。

4.2 协程模型



(a) 协程模型

图 a 展示了我们 raftcore 里面的协程模型。当应用层提案 (propose) 到来之后，在 Leader 节点，主协程会在本地追加日志，然后发送 BroadcastAppend，然后到 Follower 节点复制日志的协程会被唤醒，开始进行一轮的的复制。在 replicateOneRound 成功复制半数节点日志之后会触发 commit，rf.applyCond.Signal() 会唤醒等待做 Apply 操作的 Applier 协程。

另外 Tick 协程会监听 Timer 两个超时的 s C 通道信号。一旦心跳超时并且当前节点的状态是 Leader，Tick 就会调用 BroadcastHeartbeat 发送心

跳，发心跳的时候也会 replicateOneRound。和上述过程一样，如果半数节点成功复制日志，就会触发 commit, rf.applyCond.Signal() 就会唤醒等待做 Apply 操作的 Applier 协程。

Applier 协程 apply 完消息之后会把 ApplyMsg 消息写入 rf.applyCh 通知应用层，应用层的协程可以监听这个通道，如果有 ApplyMsg 到来就把它应用到状态机。

4.3 Rpc 定义

raft 的 rpc 定义文件在 pbs 目录下的 raftbasic.proto 文件中，主要的消息如下：

4.3.1 Entry

这个是一个日志条目信息表示。和我们之前描述的一样，它有任期号 term，索引号 index，以及操作的序列化数据 data。我们用一个字节流来存储，日志条目有两种类型一种是 Normal 正常日志，另一种是 ConfChange 配置变更的日志：

```
1
2 enum EntryType {
3     EntryNormal = 0;
4     EntryConfChange = 1;
5 }
6
7 message Entry {
8     EntryType entry_type = 1;
9     uint64    term = 2;
10    int64     index = 3;
11    bytes     data = 4;
12 }
```

4.3.2 RequestVote 相关

下面是请求投票 RPC 的定义，基本和论文里面保持一致：

请求投票里面有候选人的任期号，它的 id 以及它最后一条日志的索引以及任期号信息。

响应里面有个任期号，这个用来给候选人在选举失败的时候更新自己的任期，还有一个 vote_granted 表示这个请求投票操作是否被对端节点接受。

```
1
2 message RequestVoteRequest {
3     int64 term = 1;
4     int64 candidate_id = 2;
5     int64 last_log_index = 3;
6     int64 last_log_term = 4;
7 }
8
9 message RequestVoteResponse {
10    int64 term = 1;
11    bool vote_granted = 2;
12 }
13
14 service RaftService {
15     //...
16     rpc RequestVote (RequestVoteRequest) returns (
17         RequestVoteResponse) {}
18 }
```

4.3.3 AppendEntries 相关

日志追加操作的定义如下，基本也和论文里面一致：

请求里面有 Leader 的任期，id （用来告诉 Follower。这样 Client 访问了 Follower 之后可以被告知 Leader 节点是哪个），prev_log_index 表示消息里面将要同步的第一条日志前一条日志的索引信息，prev_log_term 是它的任期信息，leader_commit 则是 leader 的 commit 号（可以用来告知 follower 节点当前的 commit 进度），entries 表示日志条目信息。

响应里面 term 用来告诉 leader 是否出新的任期的消息，可以用来更新 leader 的任期号，success 表示日志追加操作是否成功，conflict_index 用来记录冲突日志的索引号，conflict_term 用来记录冲突日志的任期号。

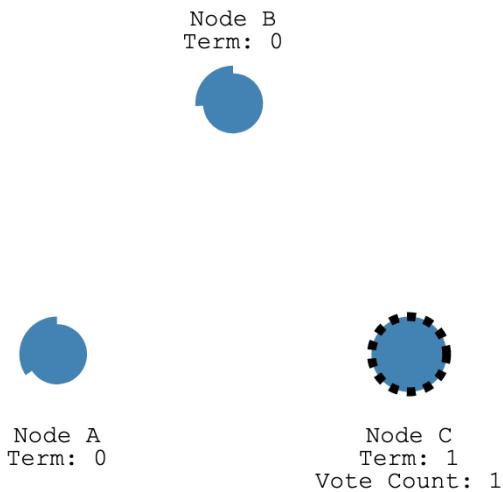
```
1
2 message AppendEntriesRequest {
3     int64    term = 1;
4     int64    leader_id = 2;
5     int64    prev_log_index = 3;
6     int64    prev_log_term = 4;
7     int64    leader_commit = 5;
8     repeated Entry entries = 6;
9 }
10
11 message AppendEntriesResponse {
12     int64 term = 1;
13     bool success = 2;
14     int64 conflict_index = 3;
15     int64 conflict_term = 4;
16 }
17
18 service RaftService {
```

```
19 //...
20 rpc AppendEntries (AppendEntriesRequest) returns (
21     AppendEntriesResponse) {}
22 }
```

4.4 Leader 选举实现分析

Raft 官网提供了一个算法的动态演示动画，我们先来直观感受下 Leader 选举的流程，然后结合代码介绍这个流程。

Raft 算法中有两个超时时间用来控制着 Leader 选举的流程。首先是选举超时，这个是 Candidate 等待变成 Leader 的时间跨度，如果在这个时间内还没被选成 Leader，这个超时定时器会被重置。我们前面也介绍过，这个超时时间设置一般在 150ms 到 300ms 之间。



(b) 选举 1

启动的时候，所有节点的选举超时时间都被设置到 $150 \sim 300ms$ 之间的随机值。那么大概率有一个节点会率先达到超时时间，如图 A,B,C 节点的 C 先达到超时时间，它从 Follower 变成 Candidate,. 随后开始新任期的选举，它会给自己投一票，然后向集群中的其他节点发送 RequestVoteRequest rpc 请求它们的投票。

1.raft 代码中在启动时，也就是应用层调用 MakeRaft 函数的时候，会传入 baseElectionTimeOutMs 和 heartbeatTimeOutMs。这里心跳超时时间是固定的。我们使用 MakeAnRandomElectonTimeout 构造生成了一个随机的选举超时时间。

```

1
2 func MakeRaft(peers []*RaftClientEnd, me int, newdbEng
```

```

storage_eng.KvStore, applyCh chan *pb.ApplyMsg,
heartbeatTimeOutMs uint64, baseElectionTimeOutMs uint64) *  

Raft {  

3  

4 ...  

5  

6 heartbeatTimer: time.NewTimer(time.Millisecond * time.Duration(  

    heartbeatTimeOutMs)),  

7 electionTimer: time.NewTimer(time.Millisecond * time.Duration(  

    MakeAnRandomElectionTimeout(int(baseElectionTimeOutMs)))),  

8 ...

```

2. 达到选举超时后, 节点 C 首先把自己状态改成 Candidate, 然后增加自己的任期号, 开始选举。

```

1  

2 //  

3 // Tick raft heart, this ticket trigger raft main flow running  

4 //  

5 func (rf *Raft) Tick() {  

6     for !rf.IsKilled() {  

7         select {  

8             case <-rf.electionTimer.C:  

9                 {  

10                     rf.SwitchRaftNodeRole(  

11                         NodeRoleCandidate)  

12                     rf.IncrCurrentTerm()  

13                     rf.Election()  

14                     rf.electionTimer.Reset(time.  

15                         Millisecond * time.Duration(  

16

```

```

        MakeAnRandomElectionTimeout(int
            (rf.baseElecTimeout)))))

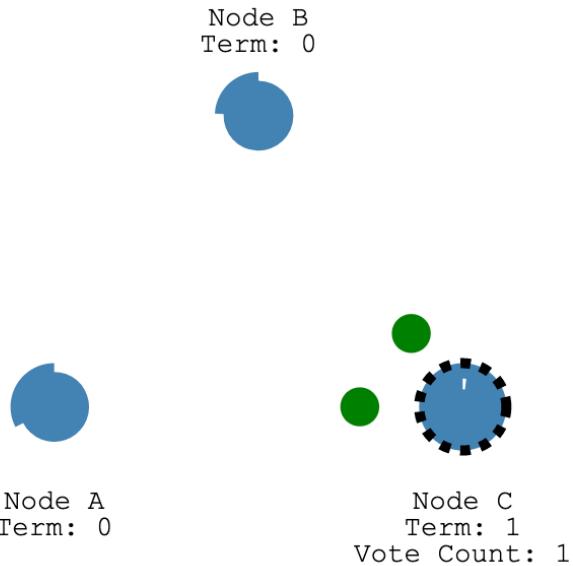
14         }
15     ...
16 }
17 }
```

3. 下面这段代码就是发起选举的核心逻辑。首先节点 IncrGrantedVotes 给自己投一票, 然后把 votedFor 设置成自己。之后构造 RequestVoteRequest rpc 请求, 带上自己的任期号, CandidateId 也就是自己的 id, 最后一个日志条目的索引和最后一个日志条目的任期号。然后把当前 Raft 状态持久化, 向集群中的其他节点并行的发送 RequestVote 请求。

```

1
2 // 
3 // Election make a new election
4 //
5 func (rf *Raft) Election() {
6     fmt.Printf("%d start election \n", rf.me_)
7     rf.IncrGrantedVotes()
8     rf.votedFor = int64(rf.me_)
9     voteReq := &pb.RequestVoteRequest{
10         Term:          rf.curTerm,
11         CandidateId: int64(rf.me_),
12         LastLogIndex: int64(rf.logs.GetLast().Index),
13         LastLogTerm:  int64(rf.logs.GetLast().Term),
14     }
15     rf.PersistRaftState()
16     for _, peer := range rf.peers {
17         if int(peer.id) == rf.me_ {
```

```
18         continue
19     }
20     go func(peer *RaftClientEnd) {
21         PrintDebugLog(fmt.Sprintf("send request
22             vote to %s %s\n", peer.addr, voteReq.
23             String()))
24
25         requestVoteResp, err := (*peer.
26             raftServiceCli).RequestVote(context.
27             Background(), voteReq)
28
29         if err != nil {
30             PrintDebugLog(fmt.Sprintf("send
31                 request vote to %s failed %v\n
32                 ", peer.addr, err.Error()))
33         }
34     }
35 }
```



(c) 选举 2

如果 A,B 收到请求的时，还没有发出投票（因为它们还没达到选举超时时间）。它们就会给候选人节点 C 投票，同时重设自己的选举超时定时器。

4. erraft 处理投票请求的细节如下，我们结合图中的例子分析下面的逻辑，假设 A 节点正在处理来自 C 的投票请求，那么首先 C 的任期号大于 A 的，代码中 1 的 if 分支不会执行。在 2 这里，A 节点发现来自 C 的请求投票消息的任期号大于自己的，它会调用 SwitchRaftNodeRole 变成 Follower 节点。在回复 C 消息之前，代码中 3 号位置 A 调用 electionTimer.Reset 重设了自己的选举超时定时器，最后再将票投给 C 节点。

```

1
2  //
```

```

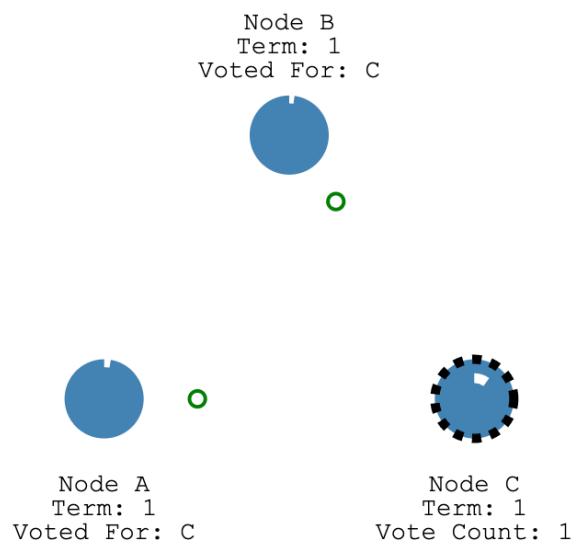
3 // HandleRequestVote handle request vote from other node
4 //
5 func (rf *Raft) HandleRequestVote(req *pb.RequestVoteRequest,
6         resp *pb.RequestVoteResponse) {
7     rf.mu.Lock()
8     defer rf.mu.Unlock()
9     defer rf.PersistRaftState()
10    //
11    if req.Term < rf.curTerm || (req.Term == rf.curTerm &&
12        rf.votedFor != -1 && rf.votedFor != req.CandidateId)
13    {
14        resp.Term, resp.VoteGranted = rf.curTerm, false
15        return
16    }
17    //
18    if req.Term > rf.curTerm {
19        rf.SwitchRaftNodeRole(NodeRoleFollower)
20        rf.curTerm, rf.votedFor = req.Term, -1
21    }
22    ...
23
24    rf.votedFor = req.CandidateId
25
26    //
27    rf.electionTimer.Reset(time.Millisecond * time.Duration(
28        MakeAnRandomElectionTimeout(int(rf.baseElecTimeout)))

```

```

28         ))
29     resp.Term, resp.VoteGranted = rf.curTerm, true
}

```



(d) 选举 3

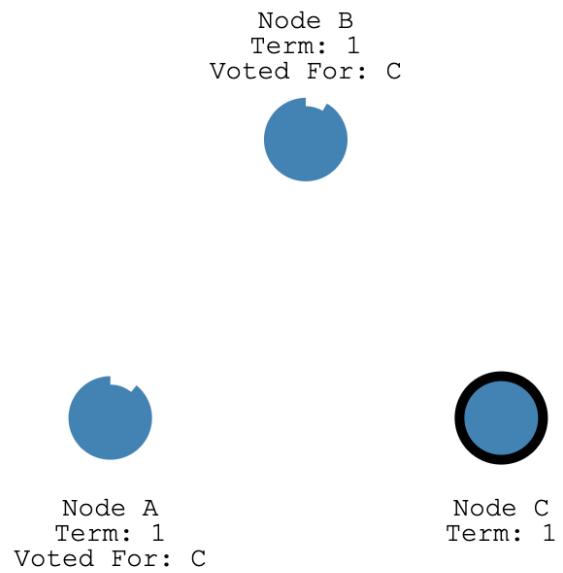
C 收到半数票以上，也就是 A, B 节点的任意一个的投票加上自己那一张选票，它就变成了 Leader，然后停止自己的选举超时定时器。

5.C 统计票数处理请求投票的响应如下，注意：这段代码加了锁，应为这里涉及到多个 Goroutine 去修改 rf 中的非原子变量，如果不加锁可能会导致逻辑错误。代码中 1 处，如果收到投票的响应 VoteGranted 是 true。C 就会调用 IncrGrantedVotes 递增自己拥有的票数，然后 if rf.grantedVotes > len(rf.peers)/2 判断是否拿到了半数以上票，如果是的调用 SwitchRaftNodeRole 切换自己的状态为 Leader。之后 BroadcastHeartbeat 广播心跳消

息，并重新设置自己的得票数 grantedVotes 为 0。

```
1 if requestVoteResp != nil {
2     rf.mu.Lock()
3     defer rf.mu.Unlock()
4     PrintDebugLog(fmt.Sprintf("send request vote to %s
5         receive -> %s, curterm %d, req term %d", peer.addr,
6         requestVoteResp.String(), rf.curTerm, voteReq.Term))
7     if rf.curTerm == voteReq.Term && rf.role ==
8         NodeRoleCandidate {
9         // 1
10        if requestVoteResp.VoteGranted {
11            // success granted the votes
12            PrintDebugLog("I grant vote")
13            rf.IncrGrantedVotes()
14            if rf.grantedVotes > len(rf.peers)/2 {
15                PrintDebugLog(fmt.Sprintf("node %d
16                    get majority votes int term %d
17                    ", rf.me_, rf.curTerm))
18                rf.SwitchRaftNodeRole(
19                    NodeRoleLeader)
20                rf.BroadcastHeartbeat()
21                rf.grantedVotes = 0
22            }
23        }
24        // 2
25    } else if requestVoteResp.Term > rf.curTerm {
26        // request vote reject
27        rf.SwitchRaftNodeRole(NodeRoleFollower)
28        rf.curTerm, rf.votedFor = requestVoteResp.
```

```
22             Term, -1  
23         rf.PersistRaftState()  
24     }  
25 }
```

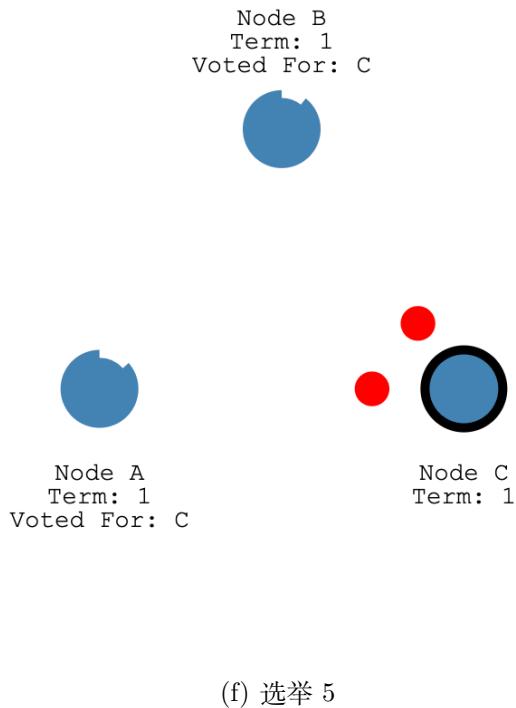


(e) 选举 4

我们知道 A,B 在前面处理投票请求的时候，只是重设的超时定时器，那么万一再一次超时定时器到达，会不会重新出发选举，然后陷入选举循环呢？

答案是不会的，我们前面只介绍了选举超时时间。还有一个心跳超时

时间，这个超时时间比选举超时时间短，一般是选举超时时间的 $1/3$ 。也就是说在 A, B 还没到达选举超时时间之前，这个心跳超时时间会先到达。如果是 Leader 节点的话，在到达心跳超时时间后，会给集群中其他节点发送心跳包，其他节点 (A, B) 接受到心跳包之后，又会重设自己的选举超时定时器。也就是说，只要 Leader C 一直正常发送心跳包，那么 A,B 节点不可能触发选举。只有当 Leader C 挂了后，A,B 节点才会开始下一轮选举。



(f) 选举 5

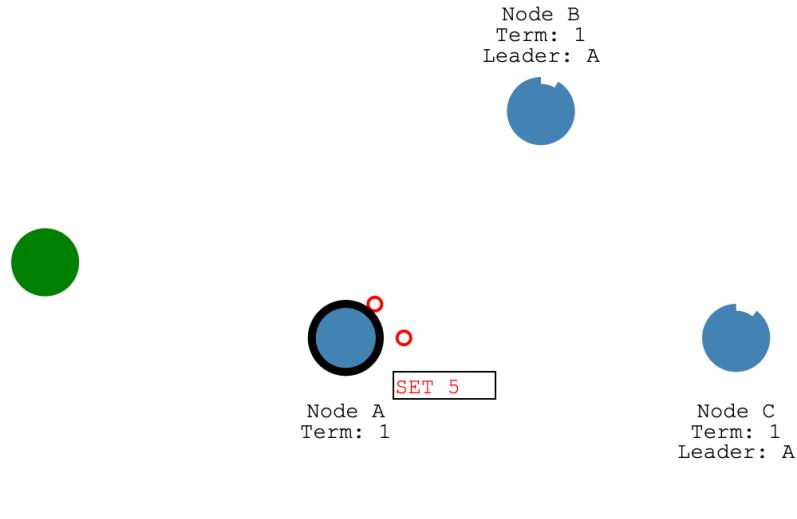
4.5 日志复制实现分析

经过上述的选举流程，我们现在就有一个拥有主节点 (Leader 节点) 和多个从节点 (Follower 节点) 的系统了。主节点会不断地给从节点发送心跳消息。现在我们要考虑如何处理客户端请求了：客户端发送一个操作过来，我们地系统是如何处理的？

首先，Raft 规定只有 Leader 节点能处理写入请求，客户端发送的请求首先会到达 Leader 节点。

在 raft 库中用户请求到来和 raft 交互的入口函数是 Propose。Propose 函数首先会查询当前节点状态。只有 Leader 节点才能处理提案 (propose)。Leader 节点会序列化用户操作，然后调用 Append 函数将操作追加到自己的日志中。之后 BroadcastAppend 将日志内容发送给集群中的 Follower 节点。

```
1 //  
2 // Propose the interface to the application propose a  
3 // operation  
4  
5 func (rf *Raft) Propose(payload []byte) (int, int, bool) {  
6     rf.mu.Lock()  
7     defer rf.mu.Unlock()  
8     if rf.role != NodeRoleLeader {  
9         return -1, -1, false  
10    }  
11    newLog := rf.Append(payload)  
12    rf.BroadcastAppend()  
13    return int(newLog.Index), int(newLog.Term), true  
14 }
```



(g) 日志复制 1

如图 g 中，绿色的表示客户端节点，它发送 SET 5 的请求。A 作为当前集群中的 Leader 节点首先会把这个 SET 5 操作封装成一个日志条目，写入到自己的日志存储结构中，然后在下一次给从节点发送心跳消息的时候带上这个日志发送给 Follower 节点，如图 h 所示。

在 erraft 实现中，我们专门有一组 Goroutine 做日志复制相关的操作。用户提案到达 Leader 之后，Leader 节点调用 BroadcastAppend 函数唤醒做日志复制操作的 Goroutine。我们使用 replicatorCond 信号量来完成 Goroutine 之间的同步操作。

```

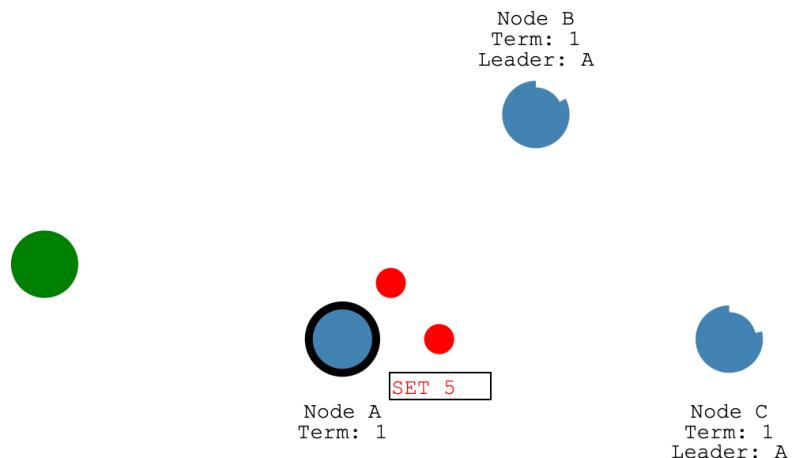
1
2 func (rf *Raft) BroadcastAppend() {
3     for _, peer := range rf.peers {
4         if peer.id == uint64(rf.me_) {
5             continue
6         }
7     }
8 }
```

```
7         rf.replicatorCond[peer.id].Signal()
8     }
9 }
```

复制操作的 Goroutine 执行的任务函数是 Replicator。当 BroadcastAppend 中通过 Signal 函数唤醒信号量，rf.replicatorCond[peer.id].Wait() 就会停止阻塞，继续往下执行，调用 replicateOneRound 进行数据复制。

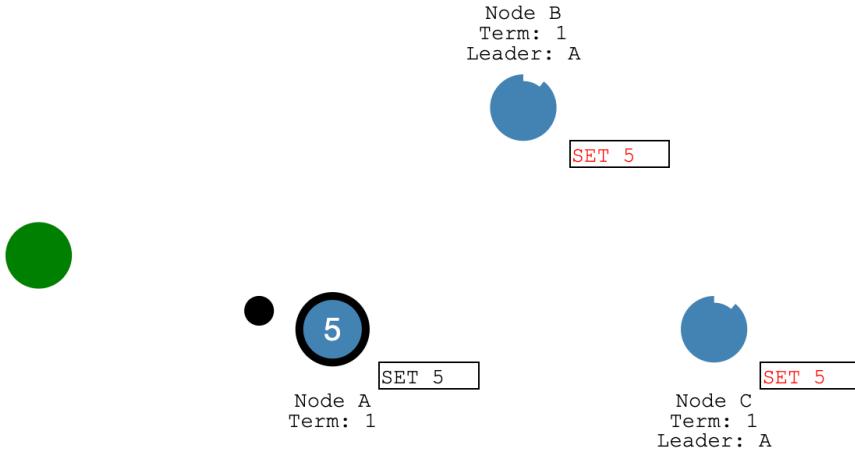
```
1
2 // 
3 // Replicator manager duplicate run
4 //
5 func (rf *Raft) Replicator(peer *RaftClientEnd) {
6     rf.replicatorCond[peer.id].L.Lock()
7     defer rf.replicatorCond[peer.id].L.Unlock()
8     for !rf.IsKilled() {
9         PrintDebugLog("peer id wait for replicating...")
10        for !(rf.role == NodeRoleLeader && rf.matchIdx[
11            peer.id] < int(rf.logs.GetLast().Index)) {
12            rf.replicatorCond[peer.id].Wait()
13        }
14        rf.replicateOneRound(peer)
15    }
}
```

replicateOneRound 就会把日志打包到一个 AppendEntriesRequest 中发送到 Follower 节点了。



(h) 日志复制 2

Follower 收到追加请求后，会把日志条目追加到自己的日志存储结构中，然后给 Leader 发送成功追加的响应。Leader 统计到集群半数节点（包括自己）日志追加成功之后，它会把这条日志状态设置为已经提交 (committed)，然后将操作结果发送给客户端，如图 i 所示，A 设置 SET 5



(i) 日志复制 3

之后这个日志提交的信息会在下一次给 Follower 发送的心跳包中发送过去。Follower 收到日志提交信息后，也会更新自己的日志提交状态。

日志提交之后，Apply 协程会收到通知，开始将已经提交的日志 Apply 到状态机中。日志成功 Apply 之后，Leader 节点会给客户端发送成功写入的响应包。

对应 etraft 实现中，日志提交之后 Leader 节点会调用 `advanceCommitIndexForLeader` 函数。它会计算当前日志提交的索引号，然后和之前已经提交的 `commitIdx` 进行对比。如果更大，就会更新 `commitIdx`，同时调用 `rf.applyCond.Signal()` 唤醒做 Apply 操作的 Goroutine。Applier 函数是 Apply Goroutine 运行的任务函数，它会调用 `Wait applyCond` 这个信号量，如果被唤醒，它会拷贝初节点中已经提交的日志，打包成 `ApplyMsg` 发送到 `applyCh` 通道通知应用层，应用层拿到 `apply` 消息之后会更新状态机并回包给客户端。

```

1
2     func (rf *Raft) advanceCommitIndexForLeader() {
3         sort.Ints(rf.matchIdx)
4         n := len(rf.matchIdx)
5         newCommitIndex := rf.matchIdx[n-(n/2+1)]
6         if newCommitIndex > int(rf.commitIdx) {
7             if rf.MatchLog(rf.curTerm, int64(newCommitIndex))
8                 {
9                     PrintDebugLog(fmt.Sprintf("peer %d advance
10                         commit index %d at term %d", rf.me_,
11                         rf.commitIdx, rf.curTerm))
12                     rf.commitIdx = int64(newCommitIndex)
13                     rf.applyCond.Signal()
14                 }
15         }
16     }
17
18     // Applier() Write the committed message to the applyCh channel
19     // and update lastApplied
20     //
21     func (rf *Raft) Applier() {
22         for !rf.IsKilled() {
23             rf.mu.Lock()
24             for rf.lastApplied >= rf.commitIdx {
25                 PrintDebugLog("applier . . .")
26                 rf.applyCond.Wait()
27             }
28             firstIndex, commitIndex, lastApplied := rf.logs.

```

```

        GetFirst().Index, rf.commitIdx, rf.
        lastApplied
27    entries := make([]*pb.Entry, commitIndex-
        lastApplied)
28    copy(entries, rf.logs.GetRange(lastApplied+1-
        int64(firstIndex), commitIndex+1-int64(
        firstIndex)))
29    rf.mu.Unlock()
30
31    PrintDebugLog(fmt.Sprintf("%d, applies entries %d
        -%d in term %d", rf.me_, rf.lastApplied,
        commitIndex, rf.curTerm))
32
33    for _, entry := range entries {
34        rf.applyCh <- &pb.ApplyMsg{
35            CommandValid: true,
36            Command:     entry.Data,
37            CommandTerm: int64(entry.Term),
38            CommandIndex: int64(entry.Index),
39        }
40    }
41
42    rf.mu.Lock()
43    rf.lastApplied = int64(Max(int(rf.lastApplied),
        int(commitIndex)))
44    rf.mu.Unlock()
45}
46}

```

4.6 Raft 快照实现分析

下面是日志快照的 RPC 定义

```
1 message InstallSnapshotRequest {
2     int64 term = 1;
3     int64 leader_id = 2;
4     int64 last_included_index = 3;
5     int64 last_included_term = 4;
6     bytes data = 5;
7 }
8
9 message InstallSnapshotResponse {
10    int64 term = 1;
11 }
12
13 rpc Snapshot (InstallSnapshotRequest) returns (
14     InstallSnapshotResponse) {}
```

InstallSnapshotRequest 中 term 代表当前发送快照的 Leader 的任期。Follower 将它与自己的任期号进行比较，来决定是否要接收这个快照。leader_id 是当前 Leader 的 id。客户端访问到 Follower 节点之后也能快速知道 Leader 信息。last_included_index 和 last_included_term 还有 data 可以参见我们第三章图 f 中的介绍，它们记录了打完快照之后第一条日志的索引号和任期号，以及状态机序列化之后的数据。

什么时间点 Raft 会打快照呢？

当日志条目过多时，我们就需要打快照。在 raft 中就是计算当前 level 中的日志条目 s.Rf.GetLogCount() 的数量来打快照的。打快照的入口函数是 takeSnapshot(index int)，传入了当前 applied 日志的 id，然后将状态机的数据序列化，调用 Raft 层的 Snapshot 函数。这个函数通过 EraseBefore-

WithDel 做了删除日志的操作，然后 PersisSnapshot 将快照中状态数据缓存到了存储引擎中。

```
1
2 // 
3 // take a snapshot
4 //
5 func (rf *Raft) Snapshot(index int, snapshot []byte) {
6     rf.mu.Lock()
7     defer rf.mu.Unlock()
8     rf.isSnapshoting = true
9     snapshotIndex := rf.logs.GetFirstLogId()
10    if index <= int(snapshotIndex) {
11        rf.isSnapshoting = false
12        PrintDebugLog("reject snapshot, current
13            snapshotIndex is larger in cur term")
14        return
15    }
16    rf.logs.EraseBeforeWithDel(int64(index) - int64(
17        snapshotIndex))
18    rf.logs.SetEntFirstData([]byte{}) // 第一个操作日志号设
19        空
20    PrintDebugLog(fmt.Sprintf("del log entry before idx %d",
21        index))
22    rf.isSnapshoting = false
23    rf.logs.PersisSnapshot(snapshot)
24}
```

什么时间点 Leader 会发送快照呢？

在复制的时候，我们会判断到 peer 的 prevLogIndex。如果比当前日志

的第一条索引号还小，就说明 Leader 已经把这条日志打到快照中了。这里我们就要构造 InstallSnapshotRequest 调用 Snapshot RPC 将快照数据发送给 Followr 节点。在收到成功响应之后，我们会更新 rf.matchIdx[peer.id]，rf.nextIdx[peer.id] 为 LastIncludedIndex 和 LastIncludedIndex + 1，更新到 Follower 节点复制进度。

```
1
2     if prevLogIndex < uint64(rf.logs.GetFirst().Index) {
3         firstLog := rf.logs.GetFirst()
4         snapShotReq := &pb.InstallSnapshotRequest{
5             Term:          rf.curTerm,
6             LeaderId:      int64(rf.me_),
7             LastIncludedIndex: firstLog.Index,
8             LastIncludedTerm: int64(firstLog.Term),
9             Data:           rf.ReadSnapshot(),
10        }
11
12        rf.mu.RUnlock()
13
14        PrintDebugLog(fmt.Sprintf("send snapshot to %s with %s\n",
15                               peer.addr, snapShotReq.String()))
16
17        snapShotResp, err := (*peer.raftServiceCli).Snapshot(
18            context.Background(), snapShotReq)
19        if err != nil {
20            PrintDebugLog(fmt.Sprintf("send snapshot to %s
failed %v\n", peer.addr, err.Error()))
21        }
22    }
23
24    if err != nil {
25        PrintDebugLog(fmt.Sprintf("apply snapshot failed %v\n", err))
26    }
27
28    if err == nil {
29        rf.matchIdx[peer.id] = rf.nextIdx[peer.id] = index
30    }
31
32    if err == nil {
33        rf.nextIdx[peer.id] = index + 1
34    }
35
36    if err == nil {
37        rf.lastIncludedIndex = index
38    }
39
40    if err == nil {
41        rf.lastIncludedTerm = firstLog.Term
42    }
43
44    if err == nil {
45        rf.logIndex = index
46    }
47
48    if err == nil {
49        rf.logTerm = firstLog.Term
50    }
51
52    if err == nil {
53        rf.logTerm = firstLog.Term
54    }
55
56    if err == nil {
57        rf.logIndex = index
58    }
59
60    if err == nil {
61        rf.logTerm = firstLog.Term
62    }
63
64    if err == nil {
65        rf.logIndex = index
66    }
67
68    if err == nil {
69        rf.logTerm = firstLog.Term
70    }
71
72    if err == nil {
73        rf.logIndex = index
74    }
75
76    if err == nil {
77        rf.logTerm = firstLog.Term
78    }
79
80    if err == nil {
81        rf.logIndex = index
82    }
83
84    if err == nil {
85        rf.logTerm = firstLog.Term
86    }
87
88    if err == nil {
89        rf.logIndex = index
90    }
91
92    if err == nil {
93        rf.logTerm = firstLog.Term
94    }
95
96    if err == nil {
97        rf.logIndex = index
98    }
99
100   if err == nil {
101      rf.logTerm = firstLog.Term
102  }
103
104  if err == nil {
105      rf.logIndex = index
106  }
107
108  if err == nil {
109      rf.logTerm = firstLog.Term
110  }
111
112  if err == nil {
113      rf.logIndex = index
114  }
115
116  if err == nil {
117      rf.logTerm = firstLog.Term
118  }
119
120  if err == nil {
121      rf.logIndex = index
122  }
123
124  if err == nil {
125      rf.logTerm = firstLog.Term
126  }
127
128  if err == nil {
129      rf.logIndex = index
130  }
131
132  if err == nil {
133      rf.logTerm = firstLog.Term
134  }
135
136  if err == nil {
137      rf.logIndex = index
138  }
139
140  if err == nil {
141      rf.logTerm = firstLog.Term
142  }
143
144  if err == nil {
145      rf.logIndex = index
146  }
147
148  if err == nil {
149      rf.logTerm = firstLog.Term
150  }
151
152  if err == nil {
153      rf.logIndex = index
154  }
155
156  if err == nil {
157      rf.logTerm = firstLog.Term
158  }
159
160  if err == nil {
161      rf.logIndex = index
162  }
163
164  if err == nil {
165      rf.logTerm = firstLog.Term
166  }
167
168  if err == nil {
169      rf.logIndex = index
170  }
171
172  if err == nil {
173      rf.logTerm = firstLog.Term
174  }
175
176  if err == nil {
177      rf.logIndex = index
178  }
179
180  if err == nil {
181      rf.logTerm = firstLog.Term
182  }
183
184  if err == nil {
185      rf.logIndex = index
186  }
187
188  if err == nil {
189      rf.logTerm = firstLog.Term
190  }
191
192  if err == nil {
193      rf.logIndex = index
194  }
195
196  if err == nil {
197      rf.logTerm = firstLog.Term
198  }
199
200  if err == nil {
201      rf.logIndex = index
202  }
203
204  if err == nil {
205      rf.logTerm = firstLog.Term
206  }
207
208  if err == nil {
209      rf.logIndex = index
210  }
211
212  if err == nil {
213      rf.logTerm = firstLog.Term
214  }
215
216  if err == nil {
217      rf.logIndex = index
218  }
219
220  if err == nil {
221      rf.logTerm = firstLog.Term
222  }
223
224  if err == nil {
225      rf.logIndex = index
226  }
227
228  if err == nil {
229      rf.logTerm = firstLog.Term
230  }
231
232  if err == nil {
233      rf.logIndex = index
234  }
235
236  if err == nil {
237      rf.logTerm = firstLog.Term
238  }
239
240  if err == nil {
241      rf.logIndex = index
242  }
243
244  if err == nil {
245      rf.logTerm = firstLog.Term
246  }
247
248  if err == nil {
249      rf.logIndex = index
250  }
251
252  if err == nil {
253      rf.logTerm = firstLog.Term
254  }
255
256  if err == nil {
257      rf.logIndex = index
258  }
259
260  if err == nil {
261      rf.logTerm = firstLog.Term
262  }
263
264  if err == nil {
265      rf.logIndex = index
266  }
267
268  if err == nil {
269      rf.logTerm = firstLog.Term
270  }
271
272  if err == nil {
273      rf.logIndex = index
274  }
275
276  if err == nil {
277      rf.logTerm = firstLog.Term
278  }
279
280  if err == nil {
281      rf.logIndex = index
282  }
283
284  if err == nil {
285      rf.logTerm = firstLog.Term
286  }
287
288  if err == nil {
289      rf.logIndex = index
290  }
291
292  if err == nil {
293      rf.logTerm = firstLog.Term
294  }
295
296  if err == nil {
297      rf.logIndex = index
298  }
299
299 }
```

```

21     rf.mu.Lock()
22     PrintDebugLog(fmt.Sprintf("send snapshot to %s with resp
23                               %s\n", peer.addr, snapShotResp.String()))
24
25     if snapShotResp != nil {
26         if rf.role == NodeRoleLeader && rf.curTerm ==
27             snapShotReq.Term {
28             if snapShotResp.Term > rf.curTerm {
29                 rf.SwitchRaftNodeRole(
30                     NodeRoleFollower)
31                 rf.curTerm = snapShotResp.Term
32                 rf.votedFor = -1
33                 rf.PersistRaftState()
34             } else {
35                 PrintDebugLog(fmt.Sprintf("set peer
36                               %d matchIdx %d\n", peer.id,
37                               snapShotReq.LastIncludedIndex))
38                 rf.matchIdx[peer.id] = int(
39                     snapShotReq.LastIncludedIndex)
40                 rf.nextIdx[peer.id] = int(
41                     snapShotReq.LastIncludedIndex)
42                     + 1
43             }
44         }
45     }
46     rf.mu.Unlock()
47 }
```

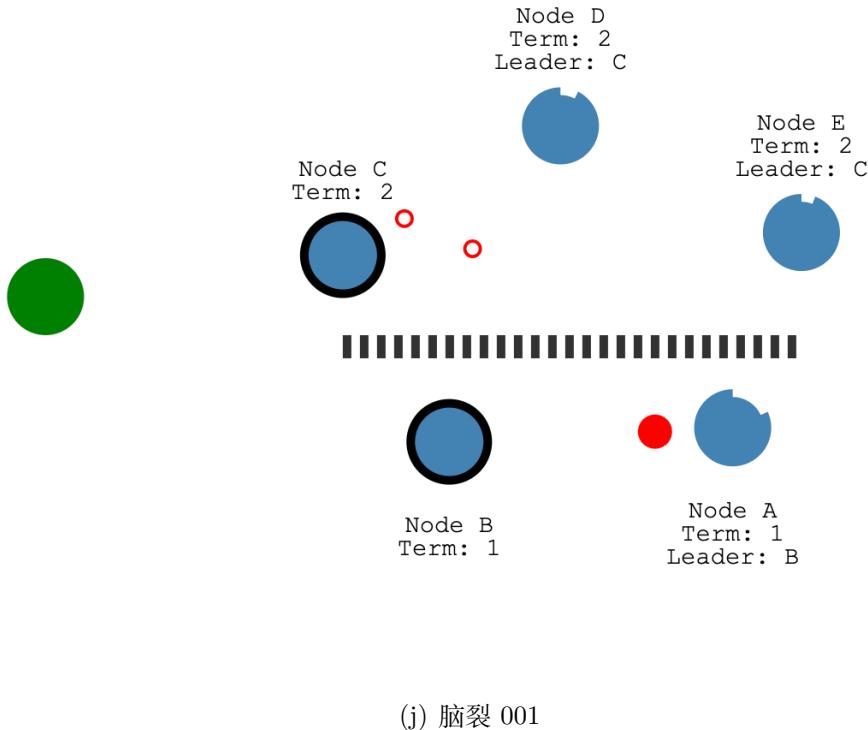
Follower 这边操作就比较简单了，它会调用 HandleInstallSnapshot 处

理快照数据，并把快照数据构造 pb.ApplyMsg 写到 rf.applyCh，负责日志 Apply 的 Goroutine 会调用 CondInstallSnapshot 安装快照，最后 restoreSnapshot 会将快照的 data 数据解析，并写入自己的状态机。

4.7 Raft 如何应对脑裂

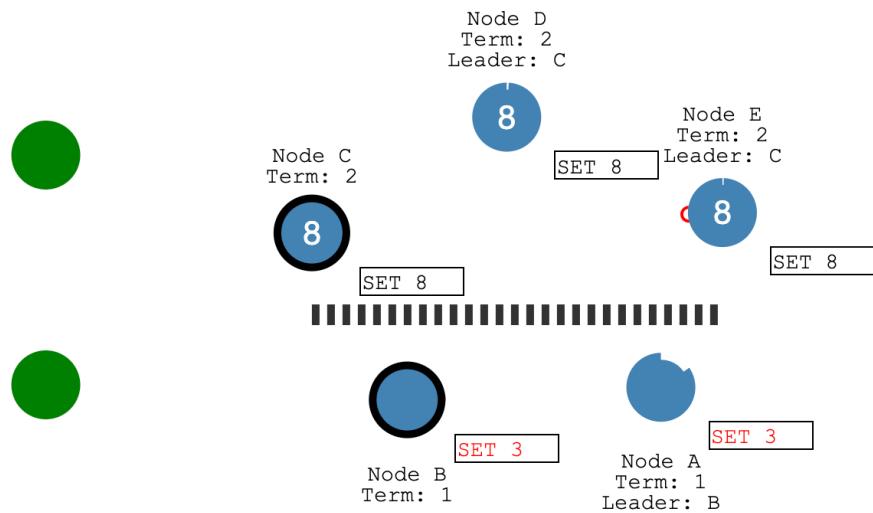
在第三章中我们，介绍了脑裂的场景，并且分析过多数派选举协议可以避免脑裂的场景，使得分布式系统在网络分区的情况下也能保持正确性。

Raft 是一种多数派选举的协议，现在我们就来看看它是如何应对脑裂的。



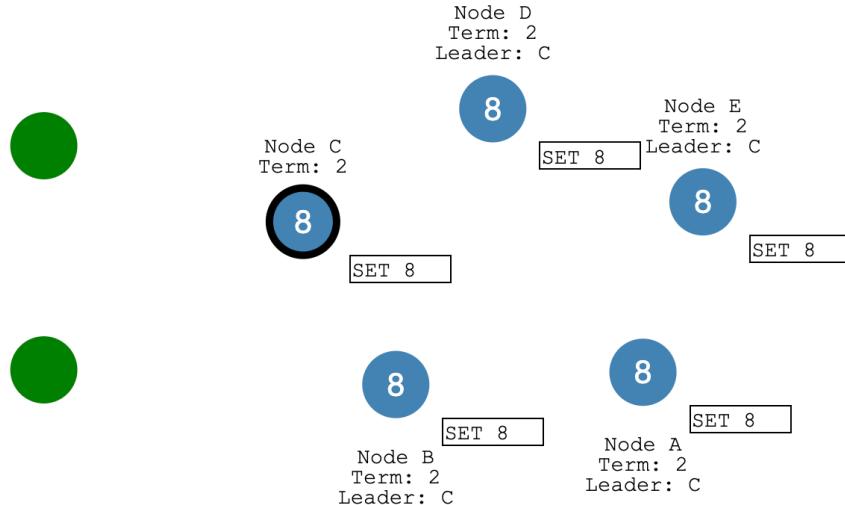
从图 j 中的场景中，我们看到一个具有 5 个节点的系统被分成了两个区：C,D,E 为一个区，A,B 为一个区。这时候两个分区中都选出了各自的

Leader，但是注意 B 是任期 1 的 Leader，C 是任期 2 的 Leader。大家可能会有疑问为什么一定有一个任期更高的 Leader，这其实也是多数派选举决定的。分区后，肯定会出现一个多数节点所在的分区，如果这个分区还没有 Leader，那么肯定会触发选举。



(k) 脑裂 002

之后如图 v 所示，C,D,E 所在分区被写入了 SET 8 的操作，由于 C,D,E 有三个节点，超过 5 个节点的半数以上，所以 SET 8 这个操作被提交了。然后 A,B 分区被写入 SET 3 操作，但是由于它们是少数派，只有两个节点，所以 SET 3 这个操作写入它们的日志之后并不能被提交。



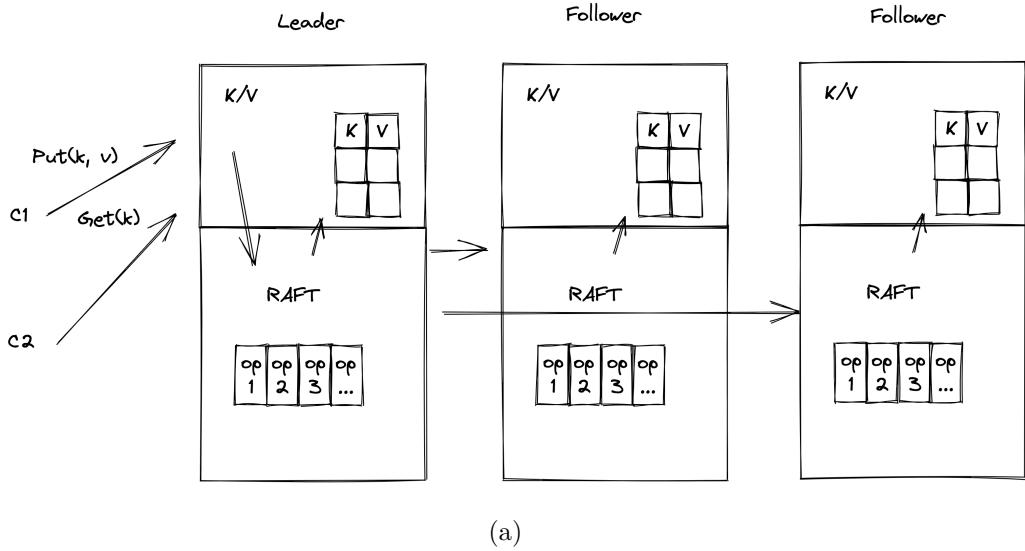
(1) 脑裂 003

最后网络恢复了，B,A 收到来自 C 的更高任期的心跳会秒变 Follower 并且会将之前没有提交的日志擦除，将 Leader C 发过来的新日志（带有 SET 8 操作）写到自己的日志中，这样整个系统仍然是一致的。

5 基于 Raft 库，实现简单的分布式 KV 系统

5.1 系统架构概览

上一章中，我们已经介绍了我们构建好的 Raft 库，它在 `raft/raftcore` 目录下。现在我们要使用这个库来构建一个高可用的分布式 KV 存储系统。



让我们回到图 a，这个图在我们一开始介绍 Raft 的时候讲到过，我们这一章就要实现这样一个系统。

5.2 对外接口定义

第一步我们来定义系统与客户端的交互接口，客户端可以发送 Put 和 Get 操作将 KV 数据写到系统中。我们需要定义这两个操作的 RPC。我们将它们合并到一个 RPC 请求里面，定义如下：

```

1 // 
2 // client op type
3 //
4 enum OpType {
5     OpPut = 0;
6     OpAppend = 1;
7     OpGet = 2;
8 }
9

```

```

10 //  

11 // client command request  

12 //  

13 message CommandRequest {  

14     string key = 1;  

15     string value = 2;  

16     OpType op_type = 3;  

17     int64 client_id = 4;  

18     int64 command_id = 5;  

19     bytes context = 6;  

20 }  

21  

22 //  

23 // client command response  

24 //  

25 message CommandResponse {  

26     string value = 1;  

27     int64 leader_id = 2;  

28     int64 err_code = 3;  

29 }  

30  

31 rpc DoCommand (CommandRequest) returns (CommandResponse) {}

```

其中 OpType 定义了我们支持的操作类型：put,Append,get。客户端请求的内容定义在了 CommandRequest 中。CommandRequest 中有我们的 key, value 以及操作类型，还有客户端 id 以及命令 id, 还有一个字节类型的 context (context 可以存放一些我们需要附加的不确定的内容)。

响应 CommandResponse 包括了返回值 value , leader_id 字段（告诉我们当前 Leader 是哪个节点的。因为最开始客户端发送请求时，不知道那

个节点被选成 Leader。我们通过一次试探请求拿到 Leader 节点的 id, 然后再次发送请求给 Leader 节点。), 以及错误码 err_code (记录可能出现的错误)。

客户端通过 DoCommand RPC 请求到我们的分布式 KV 系统。那我们的系统是怎么处理请求的呢? 首先, 我们来看看服务端的结构的定义。mu 是一把读写锁, 用来对可能出现并发冲突的操作加锁。dead 表示当前服务节点的状态, 是不是存活。Rf 比较重要, 这个是只想我们之前构建的 Raft 结构的指针。applyCh 是一个通道, 用来从我们的算法库中返回已经 apply 的日志。我们的 server 拿到这个日志之后需要 apply 到实际的状态机中。stm 就是我们的状态机了, 我们一会儿介绍。notifyChans 是一个存储客户端响应的 map, 其中值是一个通道。KvServer 在应用日志到状态机操作完之后, 会给客户端发送响应到这个通道中。stopApplyCh 用来停止我们的 Apply 操作。

5.3 服务端核心实现分析

```
1 type KvServer struct {
2     mu      sync.RWMutex
3     dead    int32
4     Rf      *raftcore.Raft
5     applyCh chan *pb.ApplyMsg
6
7     lastApplied int
8
9     stm       StateMachine
10    notifyChans map[int]chan *pb.CommandResponse
11    stopApplyCh chan interface{}
12
13    pb.UnimplementedRaftServiceServer
```

14 }

有了这个结构之后，我们要如何应用 Raft 算法库实现图中高可用的 kv 分布式系统呢？

1. 首先我们要构造到每个 server 的 rpc 客户端；
2. 然后，构造 applyCh 通道，以及构造日志存储结构；
3. 之后，调用 MakeRaft 构造我们的 Raft 算法库核心结构；
4. 最后，启动 Apply Goroutine，从通道中监听在经过 Raft 算法库之后返回的消息。

```
1 func MakeKvServer(nodeId int) *KvServer {
2     clientEnds := []*raftcore.RaftClientEnd{}
3     for id, addr := range PeersMap {
4         newEnd := raftcore.MakeRaftClientEnd(addr, uint64
5             (id))
6         clientEnds = append(clientEnds, newEnd)
7     }
8     newApplyCh := make(chan *pb.ApplyMsg)
9
10    logDbEng, err := storage_eng.MakeLevelDBKvStore("./data/
11        kv_server" + "/node_" + strconv.Itoa(nodeId))
12    if err != nil {
13        raftcore.PrintDebugLog("boot storage engine err
14            !")
15        panic(err)
16    }
17
18    // 构造 Raft 结构，传入 clientEnds，当前节点 id，日志存储的
19    db，apply 通道，心跳超时时间，和选举超时时间
20    // 由于是测试，为了方便观察选举的日志，我们设置的时间是 1s 和
```

```

    3s, 你可以设置的更短
17   newRf := raftcore.MakeRaft(clientEnds, nodeId, logDbEng,
18     newApplyCh, 1000, 3000)
19   kvSvr := &KvServer{Rf: newRf, applyCh: newApplyCh, dead:
20     0, lastApplied: 0, stm: NewMemKV(), notifyChans:
21     make(map[int]chan *pb.CommandResponse)}
22   kvSvr.stopApplyCh = make(chan interface{})}
23
24   return kvSvr
25 }
```

客户端命令到来之后，最开始调用的是 DoCommand 函数，我们来看这个函数做了哪些工作：

首先，Docommand 函数调用 Marshal 序列化了我们的 CommandResponse 到 reqBytes 的字节数组中，然后调用 Raft 库的 Propose 接口，把提案提交到我们的算法库中。Raft 算法中只有 Leader 可以处理提案。如果节点不是 Leader 我们会直接返回给客户端 ErrCodeWrongLeader 的错误码。之后就是从 getNotifyChan 拿到当前日志 id 对应的 apply 通知通道。只有这条日志通知到了，下面 select 才会继续往下走，拿到值放到 cmdResp.Value 中，当然如果操作超过了 ErrCodeExecTimeout 时间也会生成错误码，响应客户端执行超超时。

```

1
2 func (s *KvServer) DoCommand(ctx context.Context, req *pb.
3   CommandRequest) (*pb.CommandResponse, error) {
4   raftcore.PrintDebugLog(fmt.Sprintf("do cmd %s", req.
5     String())))
6 }
```

```

4
5     cmdResp := &pb.CommandResponse{}
6
7     if req != nil {
8         reqBytes, err := json.Marshal(req)
9         if err != nil {
10             return nil, err
11         }
12         idx, _, isLeader := s.Rf.Propose(reqBytes)
13         if !isLeader {
14             cmdResp.ErrCode = common.
15                 ErrCodeWrongLeader
16             return cmdResp, nil
17         }
18
19         s.mu.Lock()
20         ch := s.getNotifyChan(idx)
21         s.mu.Unlock()
22
23         select {
24             case res := <-ch:
25                 cmdResp.Value = res.Value
26             case <-time.After(ExecCmdTimeout):
27                 cmdResp.ErrCode = common.
28                     ErrCodeExecTimeout
29                     cmdResp.Value = "exec cmd timeout"
30         }
31
32     go func() {

```

```

31             s.mu.Lock()
32             delete(s.notifyChans, idx)
33             s.mu.Unlock()
34         }()
35
36     }
37
38     return cmdResp, nil
39 }
```

最后我们来看看 Apply Goroutine 干的事情：

它等待 s.applyCh 通道中 apply 消息的到来。这个 applyCh 我们在 Raft 库中提到过，它用来通知应用层日志已经提交，应用层可以把日志应用到状态机了。当 applyCh 中 appliedMsg 到来之后，我们更新了 KvServer 的 lastApplied 号，然后根据客户端的操作类型对我们的状态机做不同的操作，做完之后把响应放到 notifyChan 中，也就是 DoCommand 等待的那个通道，至此整个请求处理的流程已经结束。

```

1
2 func (s *KvServer) ApplingToStm(done <-chan interface{}) {
3     for !s.IsKilled() {
4         select {
5             case <-done:
6                 return
7             case appliedMsg := <-s.applyCh:
8                 req := &pb.CommandRequest{}
9                 if err := json.Unmarshal(appliedMsg.
10                     Command, req); err != nil {
11                     raftcore.PrintDebugLog("Unmarshal
12                         CommandRequest err")
```

```
11         continue
12     }
13     s.lastApplied = int(appliedMsg.
14                           CommandIndex)
15
16     var value string
17     switch req.OpType {
18     case pb.OpType_OpPut:
19         s.stm.Put(req.Key, req.Value)
20     case pb.OpType_OpAppend:
21         s.stm.Append(req.Key, req.Value)
22     case pb.OpType_OpGet:
23         value, _ = s.stm.Get(req.Key)
24     }
25
26     cmdResp := &pb.CommandResponse{}
27     cmdResp.Value = value
28     ch := s.getNotifyChan(int(appliedMsg.
29                           CommandIndex))
30     ch <- cmdResp
31 }
```

5.4 客户端实现介绍

客户端实现就比较简单了,主要是构造 Get 和 Put 的 CommandRequest 调用 DoCommand 发送到服务端,逻辑实现在 cmd/kvcli/kvcli.go 里面。

我们总结一下:

客户端请求到来之后，KvServer 首先会调用 Propose 提交日志到 Raft 中算法库。Raft 算法库经过共识之后提交这条日志，并通知 applyCh，KvServer 会在 Apply Goroutine 中将 applyCh 的消息解码，然后将操作应用到自己的状态机中，最后把结果写到通知客户端的 notifyChan 中，在 DoCommand 中响应结果给客户端。

6 Multi-Raft 设计与实现

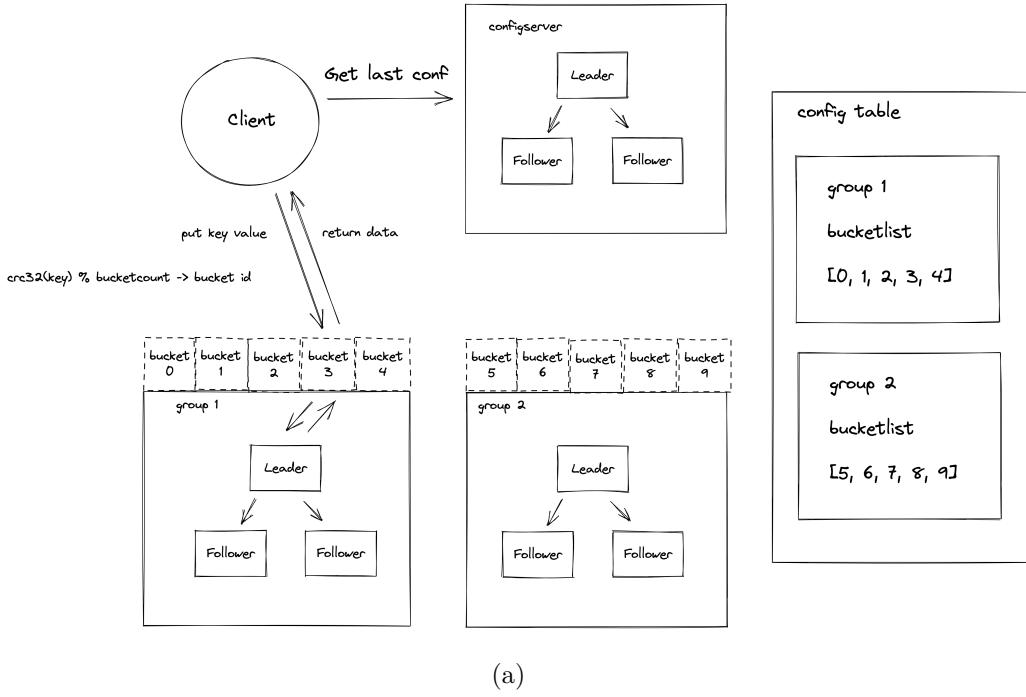
6.1 设计思考

在上一章中，我们应用 Raft 实现了一个单分组的 Raft KV 集群。客户端写请求到 Leader，Leader 把操作复制到 Follower 节点。当 Leader 挂了，会按我们第四章描述的 Raft 算法库，进行一轮新的选举，选出新的 Leader 后继续提供服务。这样我们就有了一个高可用的集群。

我们通过单分组的 KV 集群实现了高可用，以及分区容忍性，但是分布式系统还有一个可扩展的特性，也就是我们可以通过堆机器的方式来让系统实现更高的吞吐量。我们第五章实现的是单分组集群，只有单个 Leader 节点承担客户端的写入。单分组集群系统的吞吐量上线取决于单机的性能。那么我们该如何实现分布式可扩展性呢？

接下来我们将介绍 Multi-Raft 实现，它可以解决单分组集群的可扩展性问题。它的思路是这样的：既然单组有上限，那么我们可不可以多组 Raft KV 集群来实现扩展呢？我们需要有一个配置中心来管理多个分组服务器的地址信息。有了多个分组之后，我们就要考虑怎么样把用户的请求均衡地分发到相应的分组服务器上了。我们可以使用哈希算法来解决这个问题，对用户的 key 做哈希计算，然后映射到不同的服务分组上，这样可以保证流量的均衡。

整合一下上面的思路，我们可以得到如下的系统架构图 a：



(a)

在第一章开篇，我们介绍了系统的整体架构并且带大家体验了系统的运行。之后，我们讲解了 Go 语言的基础知识、Raft 算法库以及应用 Raft 算法库的示例。相信大家现在对这个系统已经有了一个更加深入的理解。

首先，客户端启动之后，会从配置服务器（ConfigServer）拉取集群的分组信息，以及分组负责的数据桶（Bucket）信息到本地。客户端发送请求的时候会计算 key 的哈希值，找到相应的桶以及负责这个桶的服务器分组（ShardServer）地址信息，然后将操作发送到对应的服务器分组进行处理。这里 ConfigServer 服务器分组和 ShardServer 服务器分组都是高可用的，多个 ShardServer 实现了系统的可扩展性。

6.2 配置服务器实现分析

配置服务的实现在 `raft/configserver` 目录下。根据上面的架构图，我们大概可以知道配置服务器需要存储哪些信息：(1) 每个服务分组的服务器地址信息；(2) 服务分组负责的数据桶信息。这个结构定义如下：

```
1
2 type Config struct {
3     Version int
4     Buckets [common.NBuckets]int
5     Groups map[int][]string
6 }
```

Version 表示当前配置的版本号。Buckets 存储了分组负责的桶信息。NBuckets 是一个常量，表示系统中最大的桶数量。这里我们默认是 10。对于大规模的分布式系统，Buckets 值可以被设置为很大。

我们看到下面的配置示例：配置版本号为 6，10 个桶和 1 个分组服务。

我们设置 127.0.0.1:6088, 127.0.0.1:6089, 127.0.0.1:6090 这三台服务器组成分组 1。Buckets 数组中 0-4 号桶都是分组 1 负责的。5-9 号桶为 0，表示当前没有分组负责这些桶的数据写入。

```
1
2 {"Version":6,"Buckets":[1,1,1,1,1,0,0,0,0,0],"Groups
   ":"{"1":["127.0.0.1:6088","127.0.0.1:6089","127.0.0.1:6090"]}}
```

raft 中把上述配置信息存储到了 leveldb 中。下面是几个操作的接口，Join 操作是把一个分组的服务器加入到集群配置中；Leave 操作是删除某些分组的服务器配置信息；Move 操作是将某个桶分配给相应的分组负责；Query 操作是查询配置信息。这些操作是通过修改 Config 这个结构，并将数据持久化到 leveldb 中实现。操作的代码实现逻辑在 raft/configserver/-config_stm.go 里面。

```
1
2 type ConfigStm interface {
3     Join(groups map[int][]string) error
```

```
4     Leave(gids []int) error
5     Move(bucket_id, gid int) error
6     Query(num int) (Config, error)
7 }
```

配置服务器的核心逻辑在 config_server.go 里面，可以看到和我们第四章实现的单分组 kv 极其类似。这里只是把 Put, Get 操作给改成了对配置的 Join、Leave、Move、Query 操作。每一次操作都经过一次共识，保证三个服务节点都有一致的配置。这样 Leader 配置节点挂掉后，集群中仍然可以从新的 Leader 配置服务器中获取配置信息。

6.3 分片服务器实现分析

首先，我们看到 bucket 定义如下：

```
1 // a bucket is a logical partition in a distributed system
2 // it has a unique id, a pointer to db engine, and status
3 //
4 type Bucket struct {
5     ID      int
6     KvDB   storage_eng.KvStore
7     Status buketStatus
8 }
```

桶具有一个唯一标识的 ID。前面我们介绍配置服务器时，介绍过一个服务分组负责一部分桶的数据。在配置服务器中，1 · 桶关联的 ID 值就是配置数组的索引号。同时，桶还关联了一个 KvStore 的接口。我们写数据的时候会传入当前服务器持有的数据存储引擎，下面是对桶中数据的操作，有 Get, Put, Append。

```
1 //  
2 // get encode key data from engine  
3 //  
4 func (bu *Bucket) Get(key string) (string, error) {  
5     encodeKey := strconv.Itoa(bu.ID) + SPLIT + key  
6     v, err := bu.KvDB.Get(encodeKey)  
7     if err != nil {  
8         return "", err  
9     }  
10    return v, nil  
11}  
12  
13 //  
14 // put key, value data to db engine  
15 //  
16 func (bu *Bucket) Put(key, value string) error {  
17     encodeKey := strconv.Itoa(bu.ID) + SPLIT + key  
18     return bu.KvDB.Put(encodeKey, value)  
19}  
20  
21 //  
22 // appned data to engine  
23 //  
24 func (bu *Bucket) Append(key, value string) error {  
25     oldV, err := bu.Get(key)  
26     if err != nil {  
27         return err  
28     }  
29     return bu.Put(key, oldV+value)
```

30 }

接下来我们来看看 ShardServer 定义的结构体

```
1 type ShardKV struct {
2     mu      sync.RWMutex
3     dead    int32
4     rf      *raftcore.Raft
5     applyCh chan *pb.ApplyMsg
6     gid_    int
7     cvCli   *configserver.CfgCli
8
9     lastApplied int
10    lastConfig configserver.Config
11    curConfig  configserver.Config
12
13    stm map[int]*Bucket
14
15    dbEng storage_eng.KvStore
16
17    notifyChans map[int]chan *pb.CommandResponse
18
19    stopApplyCh chan interface{}
20
21    pb.UnimplementedRaftServiceServer
22 }
```

这个结构和我们应用 Raft 实现单组 KvServer 的结构特别类似。这里的状态机是 map[int]*Bucket 类型的，代表当前服务器的桶的数据。分片服务器需要和配置服务器交互，知道自己负责哪些分片的数据。cvCli 定义了

到配置服务器的客户端。lastConfig, curConfig 分别记录了上一个版本以及当前版本的集群配置表。服务器知道这个表之后就知道自己负责哪些分片的数据。当集群拓扑变更后，配置表会变化，分片服务器能第一时间感知到变化，并且应用新的配置表。其他结构就和我们之前介绍单组 KvServer 一样了。

我们看看 ShardServer 构造流程和单组 KvServer 的区别。首先，Server 启动的时候我们初始化了两个引擎，一个用来存储 Raft 日志的 logDbEng，另一个用来存储实际数据的 newdbEng。cvCli 是到配置服务器分组的连接客户。, 我们调用 MakeCfgSvrClient 构造到配置服务器的客户端，传入配置服务器分组的地址列表。

```
1 //  
2 // MakeShardKVServer make a new shard kv server  
3 // peerMaps: init peer map in the raft group  
4 // nodeId: the peer's nodeId in the raft group  
5 // gid: the node's raft group id  
6 // configServerAddr: config server addr (leader addr, need to  
    optimized into config server peer map)  
7 //  
8 func MakeShardKVServer(peerMaps map[int]string, nodeId int, gid  
    int, configServerAddrs string) *ShardKV {  
9     ...  
10  
11     logDbEng := storage_eng.EngineFactory("leveldb", "./  
        log_data/shard_svr/group_"+strconv.Itoa(gid)+"/node_  
        "+strconv.Itoa(nodeId))  
12     newRf := raftcore.MakeRaft(clientEnds, nodeId, logDbEng,  
        newApplyCh, 500, 1500)  
13     newdbEng := storage_eng.EngineFactory("leveldb", "./data
```

```

14         "/group_"+strconv.Itoa(gid)+"/node_"+strconv.Itoa(
15             nodeId))
16
17     shardKv := &ShardKV{
18         ...
19         cvCli:      configserver.MakeCfgSvrClient(common.
20             UN_UNSED_TID, strings.Split(configServerAddrs
21             , ",")),
22         lastApplied: 0,
23         curConfig:  configserver.DefaultConfig(),
24         lastConfig: configserver.DefaultConfig(),
25         stm:        make(map[int]*Bucket),
26         ...
27     }
28
29     shardKv.initStm(shardKv.dbEng)
30
31     shardKv.curConfig = *shardKv.cvCli.Query(-1)
32     shardKv.lastConfig = *shardKv.cvCli.Query(-1)
33     ...
34
35     go shardKv.ConfigAction()
36
37     return shardKv
38 }
```

我们 initStm 函数初始化了状态机里面的每个 Bucket。之后，调用 cvCli.Query(-1) 查询当前最新的配置缓存到本地的 curConfig, lastConfig, 初始启动，这两个配置是一样的。

这里有一个执行任务为 ConfigAction 的 Goroutine，我们来看看它干了啥。核心逻辑如下，下面的逻辑是一个循环执行的，时间间隔是 1s。首先我们通过 cvCli.Query 尝试查询下一个配置版本信息，如果当前集群没有配置变更，返回 nil，我们 continue 进入下一轮循环，啥也不干。

如果有新的配置变更，比如加入了新的服务器分组，我们就会对比新的配置和当前配置的版本信息。如果匹配上，当前节点作为 Leader 需要把这个配置变化信息发送到这个服务器分组，让大家都知道新的配置变化。分组服务里面的每个服务器配置都要是一致的，这里我们通过 Propose 提交一个 OpType_OpConfigChange 的提案。

```
1 if _, isLeader := s.rf.GetState(); isLeader {
2 ...
3     nextConfig := s.cvCli.Query(int64(curConfVersion) + 1)
4     if nextConfig == nil {
5         continue
6     }
7     nextCfBytes, _ := json.Marshal(nextConfig)
8     curCfBytes, _ := json.Marshal(s.curConfig)
9     raftcore.PrintDebugLog("next config -> " + string(
10        nextCfBytes))
11    raftcore.PrintDebugLog("cur config -> " + string(
12        curCfBytes))
13    if nextConfig.Version == curConfVersion+1 {
14        req := &pb.CommandRequest{}
15        nextCfBytes, _ := json.Marshal(nextConfig)
16        raftcore.PrintDebugLog("can perform next conf ->
17            " + string(nextCfBytes))
18        req.Context = nextCfBytes
19        req.OpType = pb.OpType_OpConfigChange
```

```
17         reqBytes, _ := json.Marshal(req)
18         idx, _, isLeader := s.rf.Propose(reqBytes)
19         if !isLeader {
20             return
21         }
22
23         ...
24     }
25 }
```

最后我们看看分组中的服务器是怎么 Apply 这个日志的

```
1
2 nextConfig := &configserver.Config{}
3 json.Unmarshal(req.Context, nextConfig)
4 if nextConfig.Version == s.curConfig.Version+1 {
5     ...
6     s.lastConfig = s.curConfig
7     s.curConfig = *nextConfig
8     cfBytes, _ := json.Marshal(s.curConfig)
9     raftcore.PrintDebugLog("applied config to server -> " +
10                           string(cfBytes))
11 }
```

我们会更新 Server 的 lastConfig 和 curConfig 配置信息。

6.4 客户端实现分析

当客户端写入一个 Key 到系统中时，我们首先需要知道 Key 属于那个分组服务器负责。在构造客户端的时候，我们会先将最新的配置信息缓存

到本地。

```
1
2 // make a kv cilent
3 //
4 func MakeKvClient(csAddrs string) *KvClient {
5     ...
6     kvCli.config = kvCli.csCli.Query(-1)
7     return kvCli
8 }
```

客户端中我们提供了 Get(key string) 和 Put(key, value string) 的接口，它们都是调用公用的 Command 方法去访问我们的分组服务器。

```
1 //
2 // Command
3 // do user normal command
4 //
5 func (kvCli *KvClient) Command(req *pb.CommandRequest) (string,
6     error) {
7     bucket_id := common.Key2BucketID(req.Key)
8     gid := kvCli.config.Buckets[bucket_id]
9     if gid == 0 {
10         return "", errors.New("there is no shard in
11             charge of this bucket, please join the server
12             group before")
13     }
14     if servers, ok := kvCli.config.Groups[gid]; ok {
15         for _, svrAddr := range servers {
16             if kvCli.GetConnFromCache(svrAddr) == nil
```

```

14         {
15             kvCli.rpcCli = raftcore.
16                 MakeRaftClientEnd(svrAddr,
17                     common.UN_UNSED_TID)
18         } else {
19             kvCli.rpcCli = kvCli.
20                 GetConnFromCache(svrAddr)
21         }
22         resp, err := (*kvCli.rpcCli).
23             GetRaftServiceCli().DoCommand(context
24                 .Background(), req)
25         if err != nil {
26             // node down
27             raftcore.PrintDebugLog("there is a
28                 node down is cluster, but we
29                 can continue with outhter node")
30             continue
31         }
32         switch resp.ErrCode {
33         case common.ErrCodeNoErr:
34             kvCli.commandId++
35             return resp.Value, nil
36         case common.ErrCodeWrongGroup:
37             kvCli.config = kvCli.csCli.Query
38                 (-1)
39             return "", errors.New("WrongGroup")
40         case common.ErrCodeWrongLeader:
41             kvCli.rpcCli = raftcore.
42                 MakeRaftClientEnd(servers[resp.

```

```

33             LeaderId], common.UN_UNSED_TID)
34     resp, err := (*kvCli.rpcCli.
35                 GetRaftServiceCli()).DoCommand(
36                     context.Background(), req)
37     if err != nil {
38         fmt.Printf("err %s", err.
39                     Error())
40         panic(err)
41     }
42     if resp.ErrCode == common.
43         ErrCodeNoErr {
44         kvCli.commandId++
45         return resp.Value, nil
46     }
47     default:
48         return "", errors.New("unknow code
49                         ")
50     }
51 }
```

1. 首先我们会使用 Key2BucketID 函数对 Key 做 CRC32 运算，得到它应该被分配到桶的 ID；
2. 然后从本地缓存的 kvCli.config 配置里面找到负责这个 bucket id

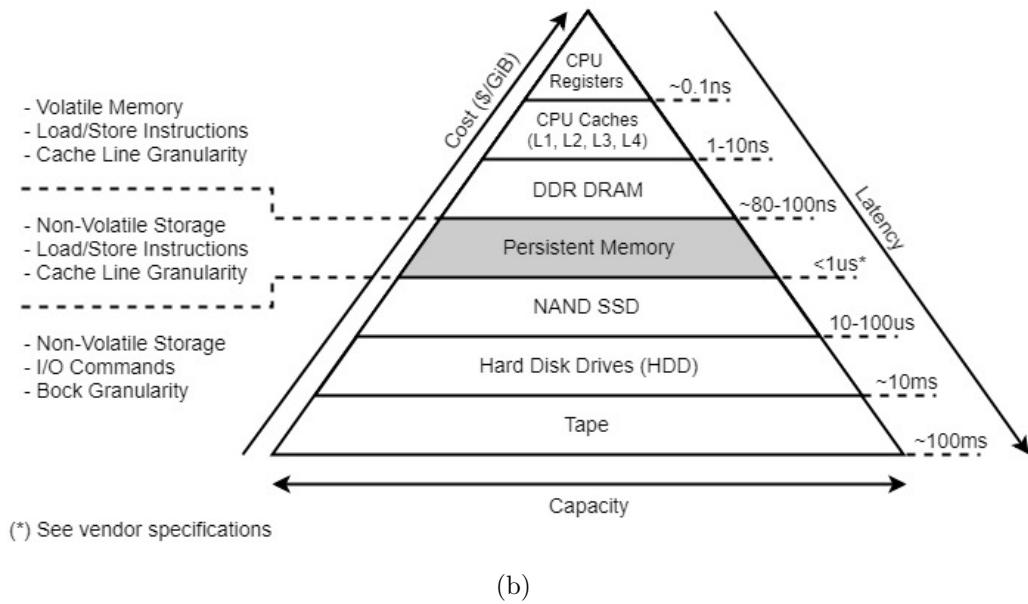
数据的服务器分组；3. 拿到服务器分组之后，我们会向第一个服务器发送 DoCommand RPC；

4. 如果这个服务器不是 Leader，它会返回 Leader 的 ID。然后客户端会重新发 DoCommand RPC 给 Leader 节点。

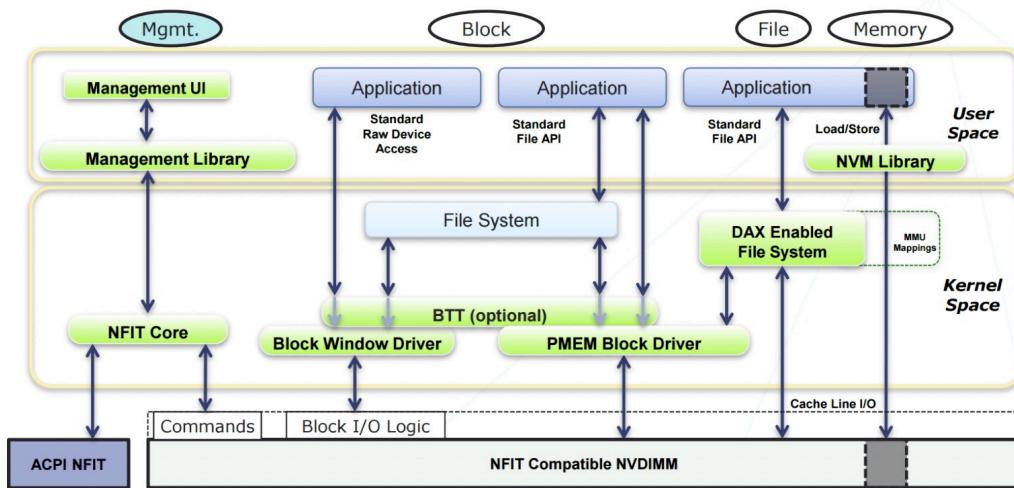
6.5 用新硬件加速持久化存储

6.5.1 持久内存介绍

Non-volatile memory (NVM) 是一种新型的计算机内存，它可以持久化的保存数据，保证断电重启后数据仍然存在，同时性能能媲美通用 DRAM 内存。



上图是持久化内存在当前分层存储体系的位置。我们可以看到它的读写速度位于 DRAM 和 SSD 之间，而且它存储的数据是持久化的。Raft 中日志和状态都需要持久化存储，我们可以把它们存储到持久内存上，获得更高的性能。



(c)

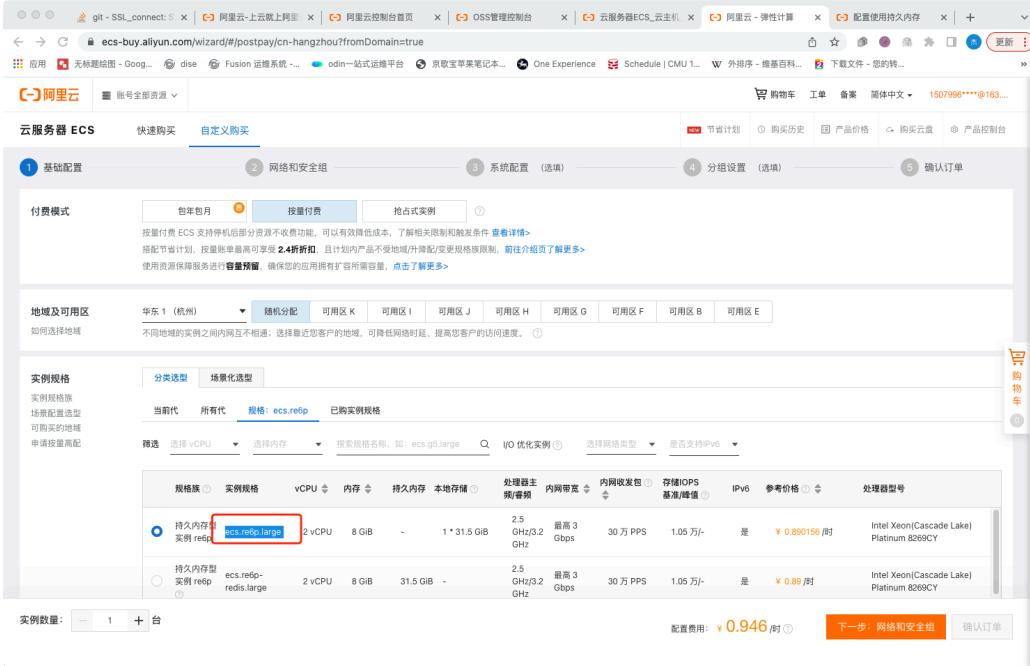
上图是 Intel 傲腾持久化内存 (AEP) 架构

它主要有下面两种工作模式：

1. Memory Mode：在这种模式下，DRAM+AEP 一起呈现为大容量内存。总容量为 AEP 的容量。DRAM 被用作 cache，对系统不可见。注意这种模式下内存是非持久性的，即断电后内容就丢失。
2. App Direct 模式：在这种模式下，AEP 呈现为 pmem 持久性内存设备（/dev/pmem）。系统看到的内存是 DRAM，应用通过操作 pmem 设备来读写 AEP。目前主流的文件系统 ext4, xfs 都支持 Direct Access 的选项（-o dax）。当挂载完并且映射完地址空间后，读写就通过 Load/Store 的内存指令来进行，绕过了传统的文件系统或者块设备访问路径。

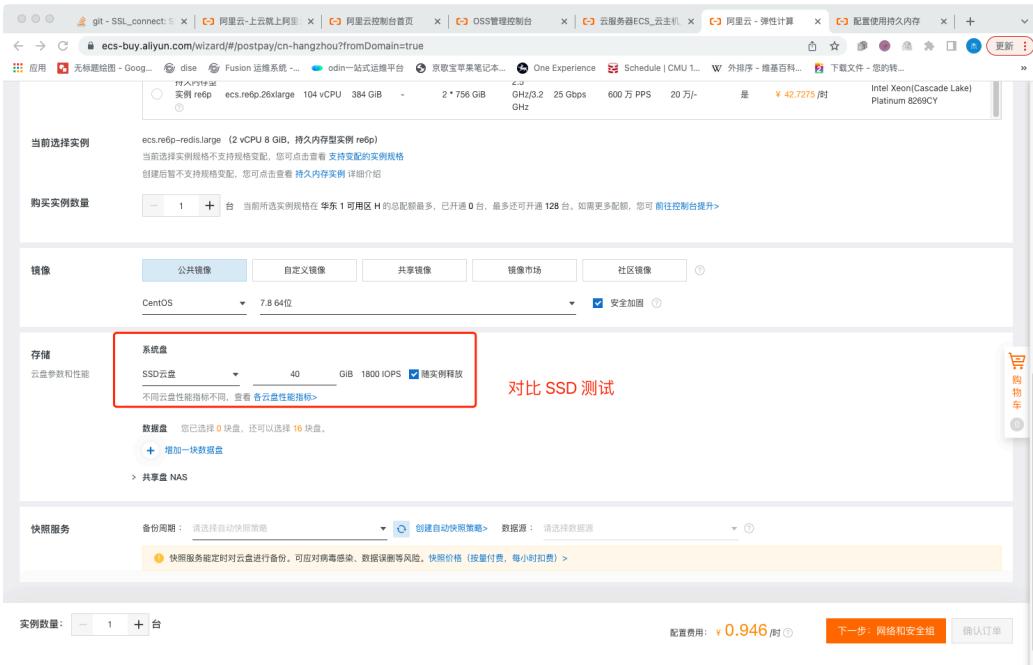
6.5.2 试用持久内存设备

目前阿里云上有按时租用的实例，可以申请 ECS 的时候规格选择
ecs.re6p.large



(d)

我们给 ECS 选上 SSD 云盘，用来给 KvRocks 做存储盘，对照性能。



(e)

配置持久化内存

我们这里将持久内存配置为一块本地盘

安装持久内存管理工具并将使用模式配置为 fsdax。

```
1 yum install -y ndctl daxctl && \
2 ndctl create-namespace -f -e namespace0.0 --mode=fsdax
```

格式化并挂载磁盘。

```
1 mkfs -t ext4 /dev/pmem0 && \
2 mkdir /mnt/sdb && \
3 mount -o dax,noatime /dev/pmem0 /mnt/sdb
```

查看已挂载的磁盘。

```
1 df -h
```

测试的库

go-redis-pmem - Golang - 持久化内存 30GiB
pmem-redis - C - 持久化内存 30GiB
kvrocks - C++ - SSD 云盘 40GiB (3000 IOPS)
redis - C - 8GiB 内存

测试环境

CPU&内存	2核(vCPU) 8 GiB	云盘	1	重新初始化云盘
操作系统	CentOS 7.8 64位	更换操作系统	快照	0
实例规格	ecs.re6p.large	更改实例规格	镜像ID	centos_7_8_x64_20... 创建自定义镜像
实例规格族	ecs.re6p		当前使用...	5Mbps (峰值) 按量付费实例更改带宽

标签

编辑标签

-

(f)

测试命令

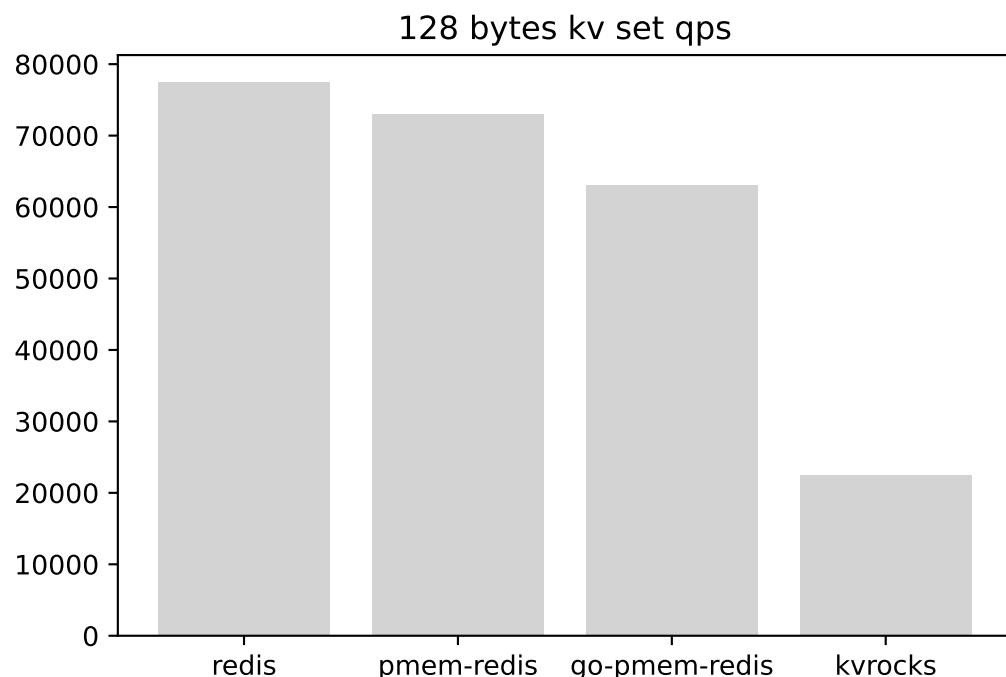
测试 GET, SET 不用 SIZE 的 key 的 qps, -d 参数设置测试 k,v 大小, 单位为 bytes

```
1 [root@iZbp1gqpip163xihy2oh56Z ~]# redis-benchmark -t set,get -d
2   128 -n 100000
3
3 [root@iZbp1gqpip163xihy2oh56Z ~]# redis-benchmark -t set,get -d
4   512 -n 100000
```

```
5 [root@iZbp1gqpip163xihy2oh56Z ~]# redis-benchmark -t set,get -d  
6 1024 -n 100000  
7 [root@iZbp1gqpip163xihy2oh56Z ~]# redis-benchmark -t set,get -d  
8 4096 -n 100000  
9 [root@iZbp1gqpip163xihy2oh56Z ~]# redis-benchmark -t set,get -d  
10 8192 -n 100000  
11 [root@iZbp1gqpip163xihy2oh56Z ~]# redis-benchmark -t set,get -d  
12 10240 -n 100000
```

测试结果

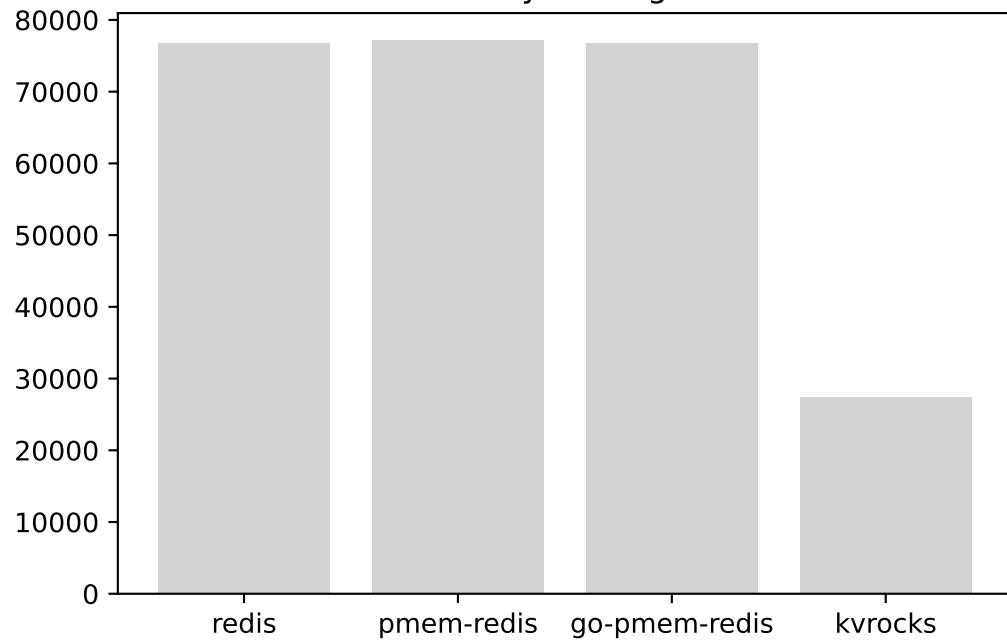
128 bytes kv set



(g)

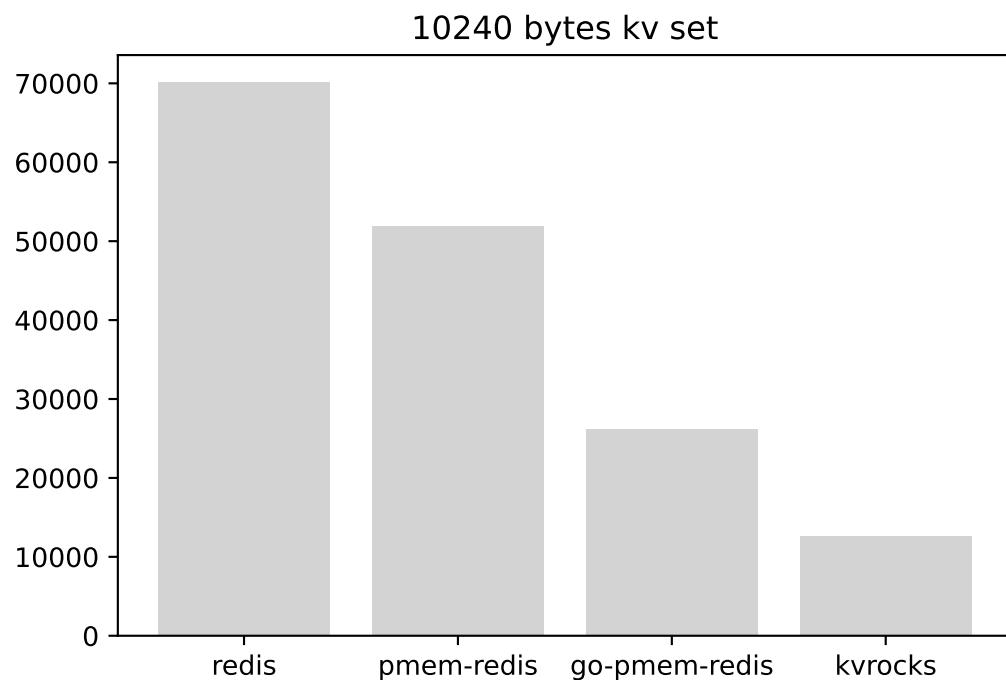
128 bytes kv get

128 bytes kv get



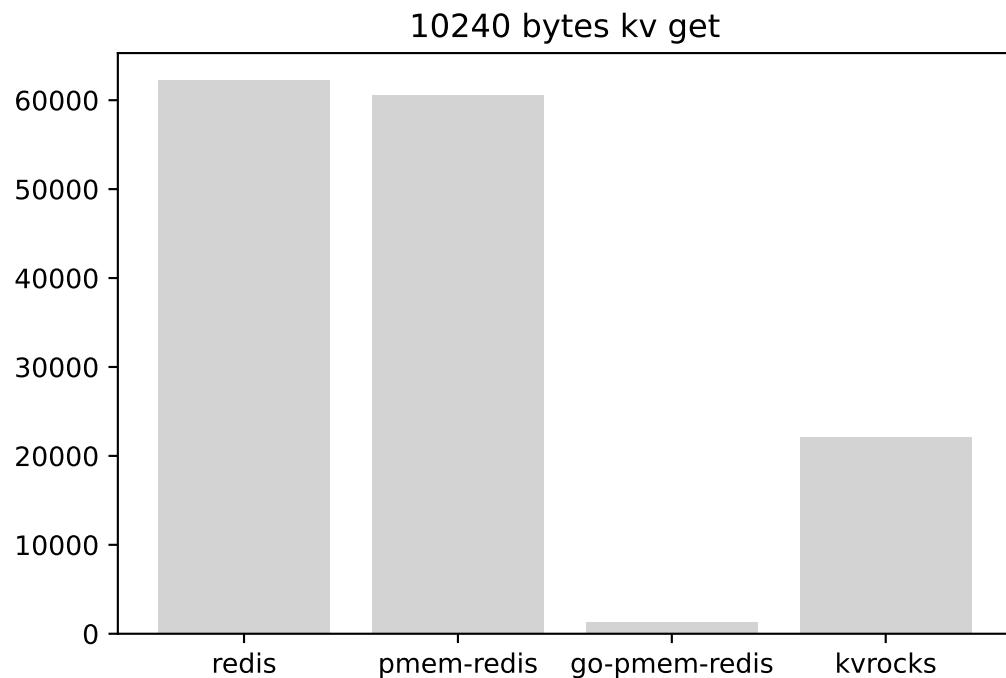
(h)

10240 bytes kv set



(i)

10240 bytes kv get

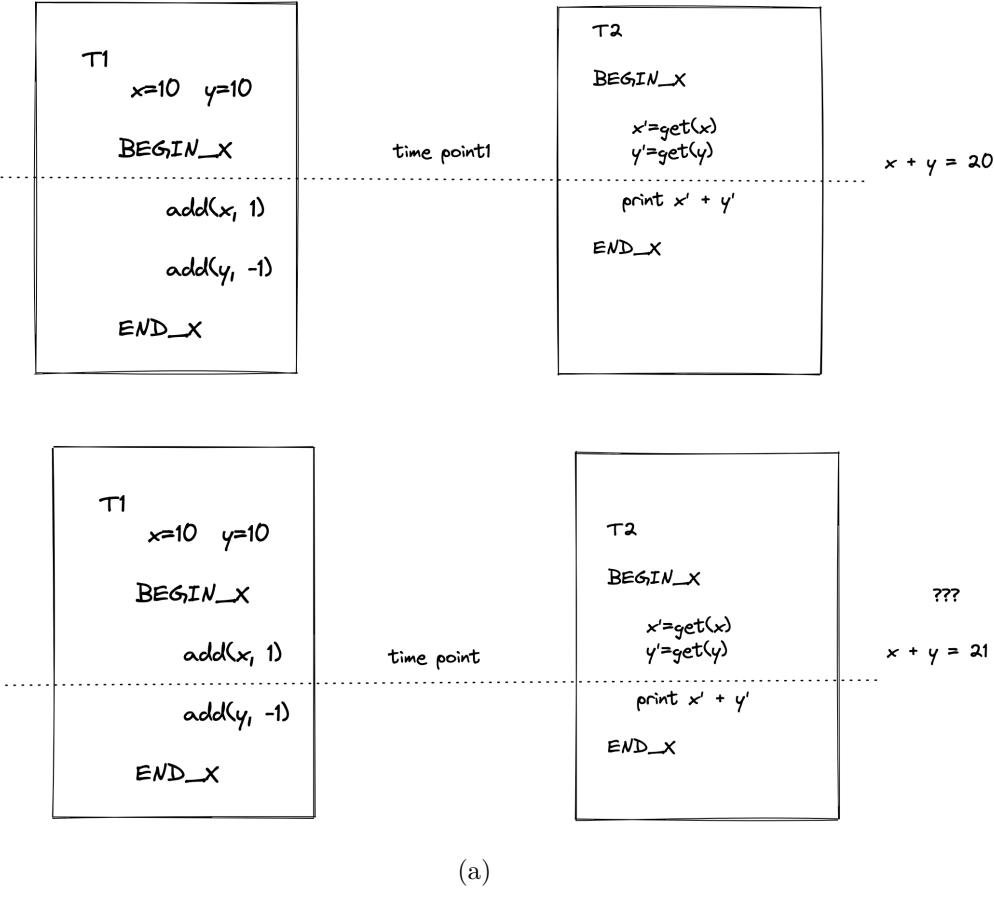


(j)

7 分布式事务初探

7.1 事务介绍

在介绍分布式事务之前，我们先来通过一个例子看看事务是什么？



开始讨论我们系统中可能发生的事情之前，我们要重新说一下事务的定义。事务是对数据库的一系列操作，这些操作满足 ACID 的属性。

我们看到上图的例子，假设我们现在实现的分布式存储系统存储了银行账户数据。

T1 表示储蓄用户的账户为 Y，他有 10 块钱，然后他给 X 转账 1 块钱，那么对应的数据操作就是对 $x + 1$ ，对 $Y - 1$ 。用户提交这个转账后，系统就开始修改数据库中的值了。

T2 表示银行对账人员，她需要统计用户 X, Y 的账户总和。如果 T2 在 T1 开始且还没有操作的时候执行 x' , y' 值得获取，那么能拿到 20 块的总和，这是符合预期的。

但是，因为两个用户使用系统的时候，他们访问的顺序是随机的，我们无法保证，一旦 T2 在 T1 执行 $\text{add}(x, 1)$ 之后读取 x', y' 的值。我们将得到 $X + Y = 21$ ，统计莫名的多出了一块钱（似乎银行亏 1 块钱也没啥问题），如果这笔转账金额很大呢，比如一个小目标 1 个亿，那就是绝对不能容忍的错误了。

这时候就需要我们的事务保障了。

7.2 ACID

- atomic 原子性。数据库管理系统保证事务是原子的，事务要么执行其所有操作，要么不执行任何操作。

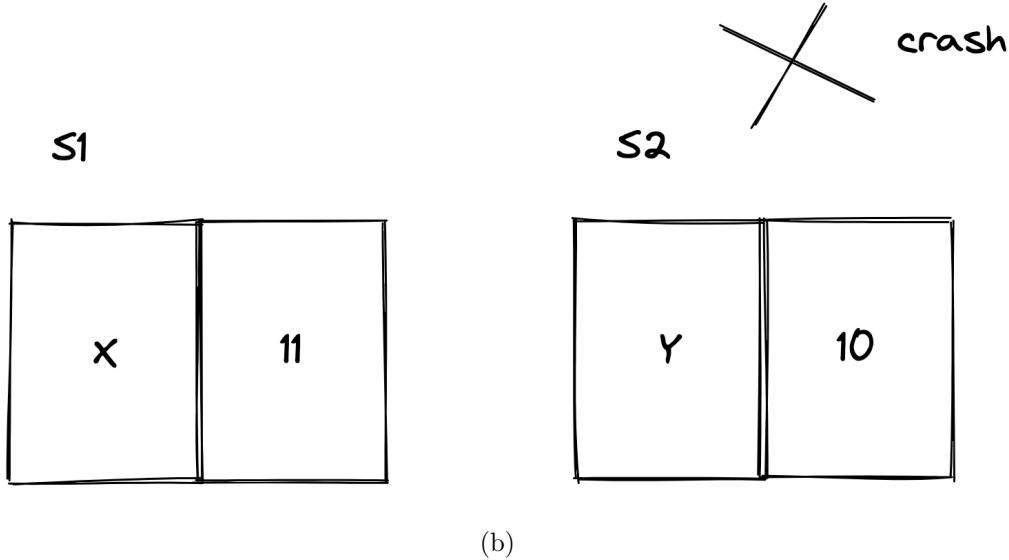
- consistent 一致性。这个表示数据库是一致的。应用程序访问的有关数据的所有查询都将返回正确的结果。

- isolated 隔离性。数据库管理系统提供了事务在系统中单独运行的假象。他们看不到并发事务的影响。这等同于事务的执行是以串行的顺序的。但是为了更好的性能，数据库管理系统必须交错并发的执行事务操作。

- durable 持久性。在系统崩溃和重启之后，提交事务的所有更改都必须是持久化的。数据库管理系统可以使用日志记录或者影子页面来确保所有的更改都是持久化的。

7.3 两阶段提交

在一个分布式系统中，数据被分割存储在不同的机器上。例如我们 erafraft 中将数据按哈希值分布到不同的 bucket，然后有不同的机器去负责这个 bucket 数据的存取。这个时候，事务处理就更复杂了。单节点我们可以通过锁保证事务正确性，但是分布式场景就不一样的，我们把上述转账示例带入分布式场景下：



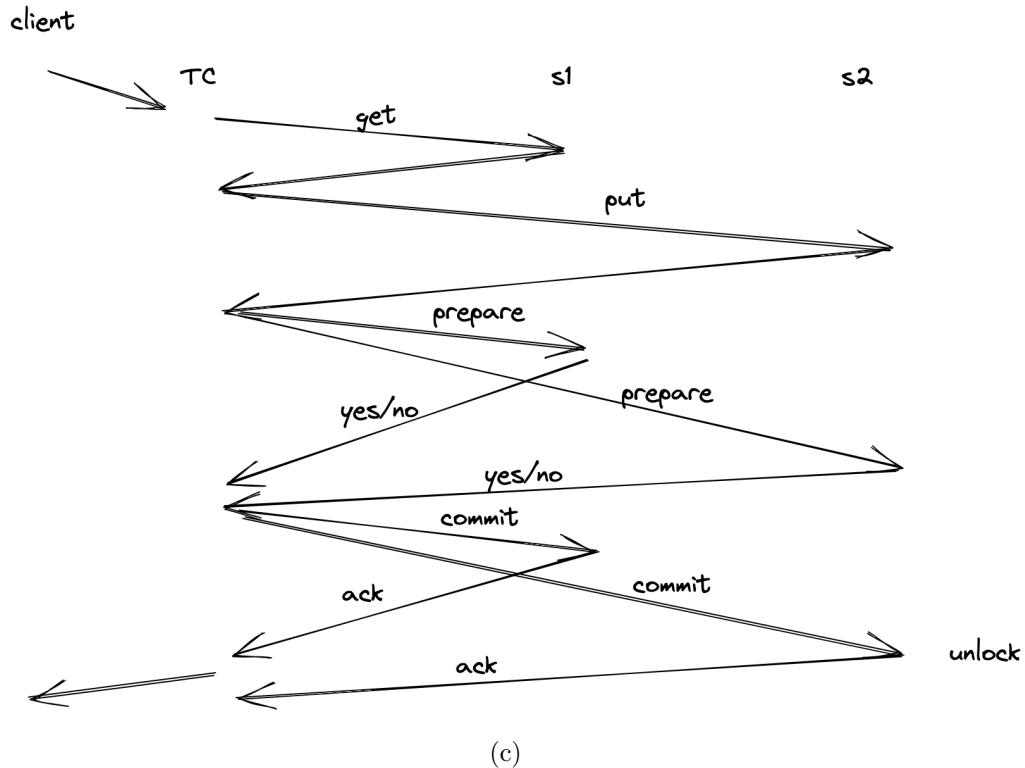
(b)

账户数据存储在 S1, S2 两台不同的机器上。T1 在 S1 上执行 +1 操作，
X 现在等于 11。当 T1 执行到对 Y 减 1 操作的时候，服务 S2 奔溃掉了。
那么这时候这个操作返回用户失败，但是 S1 上的账户已经脏了，这时候对
账人员去对账也会得到错误的数据。

面对这种场景分布式系统是如何去解决的呢？

这个时候就需要一个节点作为事务协调者（Transaction Coordinator），
来协调事务的执行了，S1, S2 负责执行事务，他们被称为事务参与者（Participants）。

我们首先概览以下两阶段提交是如何工作的



首先在我们的图中，假定 TC, S1, S2 都位于不同的服务器。TC 是事务执行的协调者。S1, S2 是持有数据的服务节点。

事务协调器 TC 会给 S1 发消息告诉它要对 X 进行 +1 操作，给服务器 S2 发消息告诉它对 Y 进行 -1 操作。后面会有一系列的消息来确认，要么 S1, S2 都成功执行了相应的，要么两个服务器都没有执行操作，不会出现非原子操作的状态，这就是两阶段提交的大致流程。